File No. 7040-21
Form C28-6335-2

IBM Systems Reference Library

IBM 7040/7044 Operating System (16/32K)

Macro Assembly Program (MAP) Language

This publication gives detailed information for writing source programs in the 7040/7044 Macro Assembly Program (MAP) language.

The MAP language is a versatile symbolic programming language that provides the user with an extensive set of pseudo-operations as well as all of the 7040/7044 machine operations.

The Macro Assembly Program, IBMAP – 7040-SP-814, is a component of the 7040/7044 (16/32K) Processor and, as such, operates under the Processor Monitor.

Preface

This publication provides the user with complete information for writing programs in the 7040/7044 Macro Assembly Program (MAP) language.

The MAP language, which encompasses all 7040/7044 machine operations, special operations, IOCS operations, prefix codes, and the pseudo-operations described herein, is the language accepted by the 16/32K 7040/7044 assembly program, IBMAP. IBMAP is a component of the 7040/7044 Processor, and, as such, operates under the Processor Monitor, using the facilities provided by the processor for input/output, loading, references to other assemblies, etc.

This publication is divided into three major parts: the first part is a description of the structure and components of the language, detailing the fundamentals of coding symbolic instructions; the second part describes in detail the MAP pseudo-operations; and the third part describes the macro-operation facility and its use. In addition, this publication contains three appendixes: a list of the 7040/7044 machine operations, special operations, IOCS operations, and prefix codes; a list of the character set acceptable to MAP; and an example of MAP assembly output. To fully describe the facilities of the MAP language, references are made to the functioning of the assembly program.

As a prerequisite to use of this publication, the reader should be familiar with the 7040/7044 Data Processing System, machine language for the 7040 7044, and the 7040/7044 Operating System, as described in the following IBM publications:

IBM 7040/7044 Systems Summary, Form A28-6289

IBM 7040/7044 Principles of Operation, Form A22-6649

IBM 7040/7044 Operating System (16/32K): Programmer's Guide, Form C28-6318

MINOR REVISION (March 1965)
This edition, Form C28-6335-2, supersedes the previous edition, Form C28-6335-1 and associated Technical Newsletters N28-0504, N28-0512, and N28-0515-0. This edition is to be included with the changes to the system released with Version 9.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the contents of this publication to:
IBM Corporation, Programming Systems Publications, Dept. D39, 1271 Avenue of the Americas, New York, N. Y. 10020

© 1963 by International Business Machines Corporation

Contents

Introduction

Communication with a computer always involves the concept of language. The programmer writes his program in a specific *source language,* which, like all languages, must conform to certain rules of structure. In general, there are three levels at which a programmer can communicate with a computer.

The first of these, and the most basic, is the language of the machine itself. Because the computer executes a job at the *machine language* level, a *source program* written at either of the other two levels must eventually be converted by some means to a machine language *object program* before it can be executed. In the 7040 and 7044, machine language consists of a sequence of binary numbers that instructs the computer in the performance of a given task. From one standpoint, the most efficient programming is done in this language, since no translation is required from source program to object program. For the programmer, however, programming in machine language is tedious, time-consuming, and given to programming errors, since, among other things, a machine language program bears little resemblance to a statement of the problem itself.

At the opposite end of the scale are such programming languages as FORTRAN and COBOL, which are scientific and commercial languages, respectively. A source program written in FORTRAN closely resembles the mathematical notation a person would use to state a problem if he were going to solve it by traditional methods. The COBOL language is based on English, and the COBOL programmer writes his source program in the form of English language statements much like those he would use to explain to someone else the procedure to be used. Both of these languages are relatively easy to learn and use because of their similarity to the ordinary languages of business and science.

Since the computer, using a compiler, can produce an efficient machine language program from a FORTRAN or COBOL source program faster and more accurately than can a programmer, the use of such *compiler languages* offers marked advantages over the use of machine language. However, certain programming features, available when using machine language, cannot be included in the scope of any present-day compiler.

At the intermediate level are the *assembly program languages,* such as the MAP language. For the most part, an assembly program language is similar in structure to machine language, but with mnemonic symbols substituted for each of the binary instruction codes and programmer symbols for the other fields of an instruction; it may also have various added features. Among these features might be a set of pseudo-operations to expand the programming facilities of machine language. Thus, the programmer has available through an assembly program language all of the flexibility and versatility of machine language, plus facilities that greatly reduce machine language programming effort.

The FORTRAN and COBOL compilers, as they are implemented in the 7040/7044 Processor, do not produce a machine language object program directly from the source program. Instead, they produce a program in the assembly program language, which is then assembled into machine language using the assembly program. To *compile,* then, is to produce assembler input from a source program written in a language such as COBOL or FORTRAN. To *assemble* is to produce a machine language program from a program either written in the assembly program language itself or produced by a compiler.

MAP Language Features

Operations

The MAP language provides the user with all of the 7040/7044 machine operations, prefix codes, special operations, and IOCS operations, as well as an extensive set of pseudo-operations.

In the MAP language, the machine operation codes are expressed in their BCD form. The prefix codes are provided to enable the programmer to specify the value of the bits in the prefix portion of a word. The special operations are extensions of standard machine operations and are provided for frequently occurring conditions. The IOCS operations are provided for the user of 7040/7044 IOCS. See Appendix A for a listing of these codes.

A pseudo-operation is any operation included in the MAP language that is not an actual 7040/7044 machine operation, prefix code, or special operation. There are more than fifty pseudo-operations available in MAP to perform the programming functions briefly described below; these functions are discussed in full later in this publication.

Location Counter Pseudo-operations are used to create and control the operation of location counters.

Data Generation Pseudo-operations are used to introduce data into the program in any one of a variety of formats. They are also used in combination to generate tables of data.

Storage Allocation Pseudo-operations are used to reserve areas of core storage.

Symbol Definition Pseudo-operations are used to assign a specific value to symbols in the program.

Program Section Pseudo-operations are used to qualify the symbols in certain sections of a program.

Literal Positioning Pseudo-operations are used to specify the location within a program where the literals used in that program are to be stored.

Macro-definition Pseudo-operations are used to define programmer macro-operations. The *macro-related pseudo-operations* are used in conjunction with these to extend the facilities of macro-operations.

Conditional Assembly Pseudo-operations are used to specify that an instruction is to be assembled only when certain criteria are satisfied.

Operation Definition Pseudo-operations are used to define new operation codes or to redefine existing ones.

Subroutine Pseudo-operations are used to generate subroutine calling sequences and to save and restore the index registers used by a subroutine.

Control Dictionary Pseudo-operations are used to make entries into the Control Dictionary to allow references between program segments. *File Description Pseudo-operations* are used to define the requirements of input/output files used by the program.

List Control Pseudo-operations are used to specify what is to be listed in an assembly output listing and what format the listing is to take.

Miscellaneous Pseudo-operations are used to specify the punching mode, to enter remarks into a program, to extend the variable field of an operation, and to indicate the end of a program.

Macro-operations

The programmer macro-operation is a very flexible and powerful programming tool. Many programming applications involve a repetition of a pattern of instructions, generally with variations in parameters at each iteration. Using the macro-definition pseudo-operations, a programmer can define this pattern as a *macro-operation,* indicating in the definition the variable parameters. Thereafter, by coding a single instruction, he can use the pattern as many times as needed, varying the parameters as much as desired. The parameters to be varied may appear in any field of any of the instructions within the pattern.

In defining the pattern, the programmer gives it a name. This name becomes the operation code of the instruction by which he will later call the macro-operation. This instruction is referred to as a *macro-instruction.* The use of macro-operations and macro-instructions is described in full in the section "Macro-operations."

Location Counters

During assembly, a *location counter* is used to determine the next location to be assigned to an instruction. For each machine instruction processed by the assembler, the location counter in effect at that time (i.e., the "current" location counter) is increased by one. Certain pseudo-operations may result in no increase at all, whereas others may result in an increase of one or more.

MAP provides for an indefinite number of symbolic location counters, which are established and controlled by the programmer. Using the location counter pseudo-operations, he can create as many location counters as he needs and then transfer control back and forth among them as often as desired.

This facility effectively means that instructions can be coded in one sequence to be loaded for execution in another, and is useful in establishing remote sequences, constant tables, etc.

ABSMOD and RELMOD Assemblies

The control routines of the operating system occupy lower core storage. Therefore, a program to be loaded may not be loaded into this area but must be loaded into the first machine location that is unused by the system. It is not necessary, however, for the programmer to know the address of this location, since the Loader will automatically *relocate* in storage each program segment to be loaded. The first address of a program segment to be executed is referred to as the *load address,* and each succeeding instruction is loaded relative to that address, so that the actual address of an instruction at load time is the address assigned to that instruction during assembly *plus* the load address of that program segment.

In its normal mode of operation, IBMAP produces an output deck, which is to be relocated at execution time by the Loader. This is referred to as a RELMOD assembly. In some cases, however, it may be desirable to load a program beginning at a certain fixed location in core storage, i.e., an absolute origin. This is referred to as an ABSMOD assembly. In this case, the programmer specifies a certain location as the load address for that deck. (An absolute origin may also be specified within a RELMOD assembly; see "Relocation Properties of Symbols.")

The basic difference between RELMOD and ABSMOD assemblies is that, in a RELMOD assembly, the relocation

bits indicate that, at load time, the contents of certain fields of the instructions may be altered relative to the load address determined by the Loader. In an ABSMOD assembly, these bits indicate that the contents of the instructions are, for the most part, not to be altered at load time; a certain amount of relocation is necessary in the case of system symbols, external symbols, and control sections.

The mode of assembly is determined by selecting either the RELMOD or the ABSMOD option on the $IBMAP card (see the 7040/7044 Programmer's Guide) that precedes the source deck.

Structure of the MAP Language

Symbolic Instructions

A symbolic instruction consists of the following three parts:

1. *The Name Field,* which may contain a name by which other instructions can refer to the instruction named. In machine instructions, use of this field is optional, and it may be left blank. Specific name field requirements for each of the pseudo-instructions are given later in this publication.

2. *The Operation Field,* which contains the machine operation code, pseudo-operation code, or programmer macro-operation code. A blank operation field will cause a word to be assembled with a prefix of zero. In this connection, note that a blank card in the symbolic program will cause a word of zeros to be generated in the object program.

An asterisk (*) may appear in the operation field immediately to the right of the last character of a machine operation code. The presence of this character indicates that the operation is indirectly addressed, and the assembler inserts the appropriate bits into the word. Indirect addressing is permitted only with certain machine operations, as shown in Appendix A.

3. *The Variable Field,* which contains the necessary address, tag, and decrement (or count) portions of a 7040/7044 machine instruction, or, in the case of a pseudo-instruction, whatever is specified for that instruction.

The address, tag, and decrement portions of the variable field are supplied in that order, separated by commas, and are referred to as *subfields* of the variable field. (Note that this order is the reverse of the internal order, from left to right, of subfields in a machine instruction word, i.e., decrement, tag, address.) Different types of instructions require different combinations of address, tag, and decrement subfields. Appendix A lists the 7040/7044 instructions, indicating for each the maximum and minimum number of subfields. The assembly program will check each instruction to see that it contains the minimum number of subfields required and that it does not contain more than the maximum number of subfields allowed. It does not check for the presence or absence of any specific subfields.

A *null subfield* is a subfield that is indicated as being present but that has no value. It is expressed in one of three ways:

a. if it is at the beginning of the variable field, by a single comma;

b. if it is between two other subfields, by two consecutive commas;

c. if it is at the end of the variable field, by a single comma followed by a blank.

If an irrelevant subfield (i.e., a subfield that is not used by the instruction) is to be followed by a relevant subfield, the irrelevant subfield must be indicated. Irrelevant subfields at the end of the variable field may be indicated as null or may be omitted entirely. Thus, the pairs of symbolic instructions below are equivalent:

```
a.      TXH     0,0,1
        TXH     ,,1
b.      IORD    ALPHA,0,1
        IORD    ALPHA,,1
c.      CLA     ALPHA,0
        CLA     ALPHA
d.      TXH     ALPHA,0,0
        TXH     ALPHA,,
e.      PXA     0,0
        PXA     ,
```

In each case above, each member of the pair is correct and neither is preferred over the other.

Note that in the last two cases it is not permissible to omit the commas, since the assembler checks for a minimum number of subfields. The TXH instruction requires three subfields; the PXA instruction, two. In other words, these subfields, although at the end of the variable field, are not irrelevant and must be indicated.

Two optional fields are available to the programmer for comments and sequencing:

1. *The Comments Field,* which is used for explanatory remarks; it exists solely for the convenience of the programmer and does not affect execution of the program.

2. *The Sequence Field,* which may be used, at the programmer's option, to indicate the sequence of cards in the source program deck. If this field is used, the

assembler will check the sequence of the cards using the 7040/7044 9-code collating sequence.

Symbolic Card Format

Symbolic instructions are punched one-per-card in the following format:

The name field begins in column 1 and may be up to six characters in length.

Column 7 is always blank, except in the case of a remarks card with an asterisk in column 1; see below.

The operation field begins in column 8 and is from one to six characters in length, plus the asterisk (if any) for indirect addressing. Therefore, it may extend through column 14.

At least one blank column separates the operation field and the variable field, which in no case may begin after column 16. The variable field may extend through column 72. Except in certain cases, the scan of the variable field of a card is terminated by a blank; if the variable field extends through column 72, the assembly program assumes a terminating blank to be present. (The blank column at the end of the variable field is actually an end-of-card indicator rather than an end-of-variable-field indicator, and it is possible to extend the variable field of most operations over several cards, using the ETC pseudo-operation.)

The comments field follows the variable field and extends through column 72. It must be preceded by at least one blank column to separate it from the variable field. In the absence of a variable field, the comments field may not begin before column 17.

Columns 73-80 are the sequence field.

REMARKS CARDS

A special type of source card is the remarks card with an asterisk (*) in column 1 and any desired information in the rest of the card. The contents of columns 1-80 are printed out in the assembly listing.

Remarks cards can be grouped, and can appear at any point in a program except between ETC cards. They are frequently used at the beginning of a program to state the problem to be solved, to describe the technique used, etc.

MAP also provides a remarks card through the pseudo-instruction REM.

Examples of Symbolic Instructions

Figure 1 illustrates the appearance of MAP symbolic instructions. The contents of the various fields will be explained in the discussions that follow.

IBM

Form X28-6333
Printed in U.S.A.

7040/44 7090/94 SYMBOLIC LANGUAGE CODING SHEET

PROGRAM		PUNCHING INSTRUCTIONS		PAGE OF
		GRAPHIC		CARD ELECTRO NUMBER
PROGRAMMER	DATE	PUNCH		

NAME (Location)	OPERATION	VARIABLE FIELD (Address, Tag, Decrement/Count)	COMMENTS	SEQUENCE (Identification)
*THE FOLLOWING		IS A PROGRAM SEGMENT WHICH ILLUSTRATES THE CODING		
*OF SYMBOLIC INSTRUCTIONS.				
	AXT	10,1		
LOOP	CLA	ALPHA+10,1	ALPHA AND BETA ARE THE SYMBOLIC ADDRESSES	
	ADD	BETA+10,1	OF THE BLOCKS OF DATA TO BE ADDED.	
	STO	GAMMA+10,1	THE SUMS ARE STORED IN BLOCK 'GAMMA.'	
	TIX	LOOP,1,1	TRANSFER IS TO SYMBOLIC ADDRESS 'LOOP.'	
	CHS	IF NO VARIABLE FIELD, COMMENTS MAY START IN COLUMN 17.		
	TRA*	X+12	THIS INSTRUCTION IS INDIRECTLY ADDRESSED.	
*ALTHOUGH THE VARIABLE FIELD MAY START BEFORE COLUMN 16, THIS COLUMN				
*IS GENERALLY ADOPTED AS THE START OF THE VARIABLE FIELD FOR ALL				
*INSTRUCTIONS. SIMILARLY, THE COMMENTS FIELD NEED NOT START IN THE SAME				
*COLUMN FOR ALL INSTRUCTIONS, BUT THIS IS GENERALLY DONE FOR EASE IN				
*READING THE ASSEMBLY LISTING.				

Figure 1. Examples of Symbolic Instructions

Symbols

A *symbol* is a string of from one to six nonblank characters. Only alphabetic characters, numeric characters, or the period may be used, and at least one character must be non-numeric.

For example, the following are valid symbols:

A	3.2
ALPHA	DECLOC
1234X	.TBL1

A symbol is said to be *defined* when it is assigned a value. As will be seen in the discussions that follow, symbol definition can occur in several ways.

ORDINARY SYMBOLS

Symbols used in the variable field of machine instructions and most pseudo-operations are of four types (exceptions for certain pseudo-operations will be noted in the discussions of those operations):

1. *Location Symbols,* which are symbols used to denote some address in the program. They are defined by their appearance in the name field of an instruction, and the value assigned is the address (relocatable or absolute) of the instruction in which they appear.

2. *Virtual Symbols,* which are symbols used in the variable field of an instruction but that never appear in the name field of an instruction; these symbols must be identified as "external symbols" and provide references to entry points in other program segments. They will then be defined at load time when the program to which they refer is loaded.

3. *Assigned Symbols,* which are symbols assigned a specific value (relocatable or absolute) by the programmer, using the symbol definition pseudo-operations.

4. *System Symbols,* which are of the form S.xxxx, where xxxx may be from zero to four characters. These symbols refer to locations defined by the operating system and the user is therefore cautioned in the use of symbols of this form. If system symbols *are* used within a program but are not defined in that program, the assembly program will automatically generate EXTERN instructions to identify them as external.

It is the programmer's responsibility in all cases to avoid circular definitions and undefined symbols.

IMMEDIATE SYMBOLS

There are some situations where it is necessary to use a symbol that is assigned a value during the first pass of the assembly program (e.g., in the variable field of an IFT operation). In MAP, normal symbol definition is not effected until the first pass of the assembler is complete. Therefore, a special class of symbols is provided whose value can be assigned and used during the first pass. These symbols, called *immediate symbols,* are assigned an arbitrary value, called the *S-value,* using the SET pseudo-operation. This value is also effective during the second pass of the assembler.

Immediate symbols can be redefined throughout the program, using additional SET pseudo-operations, and can be used in machine instructions and pseudo-instructions, subject to the following conditions:

1. An immediate symbol must be unique; it cannot have the same name as any other symbol in the program.

2. An immediate symbol may not be qualified.

3. An immediate symbol used in an instruction is given the current S-value of the symbol at the time it is encountered. Therefore, any use of immediate symbols must be preceded by a defining SET.

4. The S-value is always evaluated to yield an unsigned 15-bit integer.

For convenience in using the pseudo-instructions IFF, IFT, and SET, ordinary symbols are also assigned an S-value, as follows:

0
: if the symbol has not yet been defined as an ordinary symbol;

1
: if it has.

However, this value is effective *during the first pass only;* the actual definition of the symbol is used in the second pass.

RELOCATION PROPERTIES OF SYMBOLS

In an ABSMOD assembly, all symbols are absolute; i.e., their values are constant and will not be changed when the program is loaded for execution. In a RELMOD assembly, symbols (with the exceptions noted below) are relocatable with respect to the first location (load address) of the program, and their values are subject to change at load time.

An absolute origin may be specified within a RELMOD assembly. This is not to be confused with an ABSMOD assembly. If an absolute origin is given in a RELMOD assembly, any symbols whose definitions are dependent on that origin will be absolute (nonrelocatable). Instructions under the absolute origin may, however, refer to symbols elsewhere in the program. The assembly can be returned to the relocatable mode by subsequent specification of a relocatable origin.

Symbols that are dependent on an absolute origin may be used as entry points or as limits of a control section (described in "Control Dictionary Pseudo-Operations,") only if the assembly is ABSMOD.

The following classes of symbols are absolute symbols even though they appear within a RELMOD assembly:

1. Symbols whose definition depends on an absolute origin, i.e., as the result of an absolute ORG or an absolute BEGIN.

2. Symbols defined by:
   a. A BOOL pseudo-operation;
   b. An EQU pseudo-operation which reduces to a constant;
   c. A MAX or MIN pseudo-operation which yields a constant.

Writing Expressions

The programmer writes *expressions* to represent the subfields of the variable field of a symbolic instruction. Expressions may be used in the address, tag, and decrement portions of the variable field of a machine instruction. Expressions are also used in the variable fields of certain of the pseudo-instructions, in accordance with the rules set forth in each specific case.

The components of an expression are elements, terms, and operators.

ELEMENTS

The smallest component of an expression is the *element*. An element is either a single symbol or a single integer less than 2^{15}.

The asterisk (*) may also be used as an element. When it is used in this way, it stands for the location of the instruction in which it appears. Thus, the element * will have different values in different instructions.

TERMS

A *term* is a string composed of one or more elements and the operators

* (multiplication)
/ (division)

A term may consist of a single element, of two elements separated by an operator, of three elements separated by two operators, etc. A term must begin with an element and end with an element. It is not permissible to write two operators in succession or to write two elements in succession.

For example, the following are valid terms:

```
TMP*FUNC*TAXY
FIRST/SCND*THRD*4
3
6*2303
5*X
OFICA
```

There is no ambiguity between the use of the asterisk as an element and the use of the asterisk to denote multiplication, since the position of the asterisk always makes clear what is intended. For example, a field coded as

```
**B
```

would be interpreted as "the location of this instruction multiplied by B."

When the asterisk is used as an element, it is a relocatable symbol unless it appears under an absolute origin.

EXPRESSIONS

An *expression* is a string composed of one or more terms and the operators

\+ (addition)
− (subtraction)

Any expression may consist of a single term, of two terms separated by an operator, of three terms separated by two operators, etc. It is not permissible to write two terms in succession or two operators in succession, but an expression may begin with a + or −. For example, the following are valid expressions:

```
3
OFICA
TMP-4
-77
-TMP*FUNC/X-7*H+13759601*YMNG*ZWYT/4+3
*-A+B*C
```

RULES FOR FORMING EXPRESSIONS

Let R stand for a relocatable symbol, and A for an absolute symbol or constant. Then the following relationships hold and are acceptable to MAP:

R−R = A	R*R = complex	A+A = A
R−A = R	R*A = complex	A−A = A
A−R = complex	A/R = complex	A*A = A
R+R = complex	R/R = complex	A/A = A
R+A = R	R/A = complex	

A complex expression is one that is neither simply relocatable (i.e., R) nor absolute. A virtual symbol is a complex element. Any expression containing a complex element is itself complex.

Relocatable and complex expressions are generally evaluated at load time, when absolute values are assigned to the symbols as part of the loading process. In the case of pseudo-operations that may affect a location counter (e.g., BES and BSS) or the definition of a symbol (e.g., MAX and MIN), it is necessary that the variable field of such operations be evaluated before load time. Therefore, certain rules must be observed in using expressions; these rules will be stated in the discussions of the pseudo-operations affected.

EVALUATION OF EXPRESSIONS

Expressions are evaluated in the same order as they are described above. That is, first the elements are evaluated, then the individual terms, and finally the complete expression. The procedure is as follows:

1. Each element is replaced with its numeric value.
2. Each term is evaluated by performing the indicated multiplications and divisions from left to right. In division, the integral part of the quotient is retained and any remainder is discarded immediately. For example, the value of the term 5/2*2 is 4.

In the evaluation of a term, division by zero is the same as division by one and is not regarded as an error.

3. The terms are combined from left to right in the order in which they occur, with all intermediate results retained as 35-bit signed numbers.

4. Finally, the rightmost 15 bits are retained, complemented if the result was negative.

Grouping of terms, by parentheses or other means, is not permitted, but this is rarely a serious restriction, since a product such as A(B+C) can be written as

```
A*B+A*C
```

The expression ** is commonly used to designate a field whose value is to be computed and inserted by the program. It is an absolute expression of zero value.

RELATIVE ADDRESSING

Once an instruction has been named by the presence of a symbol in the name field, it is possible for other instructions to refer to that instruction using that symbol. Moreover, it is possible to refer to instructions preceding or following the instruction named by indicating their position relative to that instruction. This procedure is referred to as *relative addressing*. A relative address is, effectively, a type of expression.

For example, given the sequence

```
ALPHA    TRA    BETA
         CLA    GAMMA
         SUB    DELTA
STGAM    STO    GAMMA
         TPL    LOCI
```

control may be transferred to the instruction CLA GAMMA by either of the following instructions:

```
         TRA    ALPHA+1
         TRA    STGAM-2
```

Since some pseudo-operations may generate more than one word or no words in the object program, care must be exercised in using relative addressing in combination with pseudo-operations. For example, the instruction

```
ALPHA    OCT    2732,427,12716
```

generates three words of octal data in the object program, with ALPHA assigned to the address of the first word generated; therefore, the address ALPHA+2 refers to the third word generated, i.e., 12716.

It is also possible, using relative addressing, to refer to a word in a block of storage reserved by a BSS or BES pseudo-operation. For example, the instruction

```
BETA     BSS    50
```

reserves a block of 50 words, where BETA is assigned to the first word of the block. Then the address BETA+1 refers to the second word, and BETA+n refers to the $(n+1)th$ word.

Decimal Data Items

A decimal data item is used to specify in decimal form a word or words of data to be converted into binary form. Decimal data items may be used in the variable field of a DEC pseudo-instruction, or, when preceded by an equal sign, as decimal literals.

Three types of decimal data items are recognized:

1. *Decimal Integers.* A decimal integer is a string of digits, from 0 through 9, which may be preceded by a plus or minus sign. The maximum size of a decimal integer is $2^{35}-1$. For example, the decimal integer

```
-31
```

would be converted to a 35-bit signed binary number whose octal representation is

```
-000000000037
```

2. *Floating-Point Numbers.* A floating-point number has two components:

   a. The *principal part* is a signed or unsigned decimal number, which may be written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the decimal number. If the exponent part (see below) is present, the decimal point may be omitted, in which case it is assumed to be located at the right-hand end of the decimal number. The principal part cannot exceed 20 digits. If it does, the number will be truncated and only the first 20 digits will be used.

   b. The *exponent part* consists of the letters E or EE followed by a signed or unsigned decimal integer. The exponent part may be omitted if the principal part contains a decimal point. If used, it must follow the principal part. The exponent part cannot exceed two digits. If it does, it will be truncated and only the first two digits will be used.

If the letters EE are present, the number is taken as a double-precision floating-point number.

A floating-point number will be converted to a normalized floating-point binary word. The exponent part, if present, specifies a power of ten by which the principal part will be multiplied during conversion. For example, all of the following floating-point numbers are equivalent and will be converted to the same floating-point binary number:

```
3.14159
31.4159E-1
314159.E-5
314159E-5
.314159E1
```

The octal representation of this number is

```
202622077174
```

Similarly, the number .314159EE1 will be converted to a double-precision floating-point number whose octal representation is

202622077174
147015606335

3. *Fixed-Point Numbers.* A fixed-point number has three components:

a. The *principal part* is a signed or unsigned decimal number, which may be written with or without a decimal point. The decimal point may appear at the beginning, at the end, or within the decimal number. If the exponent part is present, the decimal point may be omitted, in which case it is assumed to be located at the right-hand end of the decimal number. The principal part cannot exceed 20 digits. If it does, the number will be truncated and only the first 20 digits will be used.

b. The *exponent part* consists of the letters E or EE followed by a signed or unsigned decimal integer. The exponent part may be omitted if the principal part contains a decimal point. If used, it must follow the principal part. The exponent part cannot exceed two digits. If it does, it will be truncated and only the first two digits will be used.

c. The *binary-place part* consists of the letters B or BB followed by a signed or unsigned decimal integer. The binary-place part must be present in a fixed-point number and must come after the principal part. If the number has an exponent part, the binary-place part may precede or follow the exponent part. The binary-place part cannot exceed three digits. If it does, it will truncated and only the first three digits will be used.

If the letters EE or the letters BB are present, the number is taken as double-precision.

A fixed-point number is converted to a fixed-point binary number that contains an understood binary point. The purpose of the binary-place part of the number is to specify the location of this understood binary point within the word. The number that follows the letter(s) B or BB specifies the number of binary places in the word to the left of the binary point (that is, the number of integral places in the word.) The sign bit is not counted. Thus, a binary-place part of zero specifies a 35-bit binary fraction. B2 specifies two integral places and 33 fractional places. B35 specifies a binary integer. B−2 specifies a binary point located two places to the left of the leftmost bit of the word; that is, the word would contain the low-order 35 bits of a 37-bit binary fraction. As with floating-point numbers, the exponent part, if present, specifies a power of ten by which the principal part will be multiplied during conversion.

For example, the following fixed-point numbers all specify the same configuration of bits, but not all of them specify the same location for the understood binary point:

22.5B5
11.25B4
1125E−2B4
9B7E1

All of the above fixed-point numbers will be converted to the binary configuration whose octal representation is

264000000000

The following double-precision fixed-point numbers

10BB35
1B35EE1
1BB35E1
1BB35EE1

will be converted to the binary configuration whose octal representation is

000000000012
000000000000

Literals

It is often necessary for a programmer to refer to a location containing a constant. For example, if he wishes to add the number 1 to the contents of the accumulator, he must have somewhere in storage a location containing the number 1. The introduction of data words and constants into a program is most easily accomplished using a literal. A literal is a symbol or quantity which is itself data rather than a reference to data.

In contrast to other types of subfields, the content of a literal subfield is itself the data to be operated upon. The appearance of a literal directs the assembler to prepare a constant equal in value to the content of the literal subfield; store this constant in a special table (called the *Literal Pool*); and replace the field of the instruction containing the literal with the address of the constant thus generated.

There are three types of literals: decimal, octal, and alphameric.

DECIMAL LITERALS

A decimal literal consists of an equal sign (=) followed by a decimal data item. Thus, three types of decimal literals are recognized: decimal integers, fixed-point numbers, and floating-point numbers.

A decimal literal is considered to be single-precision except where a double E or a double B is used. Double-precision literals are stored in consecutive locations,

with the high-order part first in an even location relative to the beginning of the Literal Pool.

For example, the following are valid decimal literals:

```
=1409
=31.4159E-1
=1125E-2B4
```

OCTAL LITERALS

An octal literal consists of an equal sign (=), followed by the letter O, followed by a signed or unsigned octal integer. An octal integer is a string of not more than 12 digits. The permissible characters are

```
+ - 0 1 2 3 4 5 6 7
```

For example, the following are valid octal literals:

```
=O2303
=O-12716
=O-123456712345
```

ALPHAMERIC LITERALS

An alphameric (Hollerith) literal consists of an equal sign (=), followed by the letter H, followed by exactly six alphameric characters (see Appendix B); the six characters following the H are taken as data even if one or more of them is a comma or a blank.

For example, the following are valid alphameric literals (where b represents a blank):

```
=H123456
=HTADbbb
=HXYZb,b
```

Error Checking

IBMAP incorporates a comprehensive system of error checking. Source programs are thoroughly checked for errors in the use of the language, and the line number of an instruction in error is printed out at the end of the assembly listing along with the corresponding error message(s). This enables the programmer to locate errors quickly and correct them according to the information in the error message.

Rules for Sequence Checking

If columns 73-80 of the source program cards are not blank, the assembly program will check them for sequence (using the 7040/7044 9-code collating sequence) and will issue warning messages, as follows:

Any card out of sequence will cause a warning message.

If the assembler encounters one or more duplicate sequence fields, it will issue a warning message on the first (and only the first) duplication.

Blank sequence fields are ignored and are not treated as a sequence error.

Strictly speaking, there is a distinction between an operation and an instruction, although the terms are frequently used interchangeably. The instruction is the order to the computer to perform in some manner, and the operation is the actual internal functioning of the computer. In the context of this publication, "instruction" refers to the statement that is written by the programmer.

A *pseudo-operation* might be defined as any operation available in MAP that is not an actual machine operation. Then, by extension, a *pseudo-instruction* is any valid MAP instruction that is not a machine instruction.

Note that while machine instructions generate words in the object program on a one-for-one basis, pseudo-instructions may generate no words, one word, or more than one word in the object program.

The MAP pseudo-operations (except for those related to macro-operations) are described in full in this part of the publication. They are grouped according to type of function. For each operation, there is a description of the format of the instruction and a discussion of how the operation is used.

Location Counter Pseudo-operations

Location counters provide the user with the facility to write instructions in one sequence to be loaded in another. This is useful in establishing remote sequences, etc. IBMAP assigns the necessary origins at assembly time, but repositioning of instructions under the various location counters is a function of the Loader and occurs at load time; the object deck produced by an assembly will have the same sequence as the source program deck.

The basic location counter of MAP is referred to as the *blank location counter,* since it has no associated symbol. A second MAP location counter, denoted by two slashes (//), enables the user to obtain the blank COMMON area of 7040/7044 FORTRAN IV (described in "The CONTRL Pseudo-operation").

In addition to the blank counter and the // counter, MAP provides the facility for creating and controlling as many symbolic location counters as desired. Since the symbols representing location counters are used only in the pseudo-instructions USE, BEGIN, and CONTRL, they may duplicate any other symbol in the program without causing ambiguity.

The USE Pseudo-operation

The USE (Use Location Counter) pseudo-operation specifies which of the location counters is to control the sequence of the instructions following it. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	USE	One of: 1. A single symbol (or //) 2. Blanks 3. The word PREVIOUS

The effect of this operation is to place succeeding instructions under control of the location counter represented by the contents of the variable field. The location counter in control at the time the USE is encountered is suspended at its current value, is temporarily preserved as the "previous" counter, and is continued from this value if reactivated by another USE.

If the USE PREVIOUS option is selected, the previous location counter is reactivated. For example, the sequence

```
USE     A
USE     B
USE     PREVIOUS
```

is identical in effect to

```
USE     A
USE     B
USE     A
```

The blank location counter is primary: if no USE is given, instructions will be assembled under it.

The initial value of the blank location counter is always taken as zero. The initial value of the *nth* location counter is taken as the last value reached by the $(n-1)st$ location counter, except where a BEGIN pseudo-instruction is given. Location counters are sequenced with the blank counter first, the other location counters in the order of their first appearance in a USE or BEGIN, and the // counter last.

The BEGIN Pseudo-operation

The BEGIN pseudo-operation specifies a location counter and gives it an initial value. The format of the BEGIN instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	BEGIN	Two subfields, separated by a comma: 1. A location counter symbol 2. Any non-complex expression; if it is relocatable, it must be positive

The effect of this operation is to define the symbol in the first subfield of the variable field as a location counter symbol whose initial value is the value of the expression in the second subfield. For example, if the instruction

```
BEGIN     CNTRA, ALPHA+50
```

is given, then the first appearance of a USE CNTRA instruction would cause succeeding instructions to be assembled beginning at location ALPHA+50.

A BEGIN may appear anywhere in the program, regardless of the location counter in control. No BEGIN may be given for the blank location counter.

The following example illustrates the use of location counters:

```
instruction  1
BEGIN        A, *
USE          A
instruction  2
instruction  3
BEGIN        C, *
instruction  4
instruction  5
USE          //
instruction  6
instruction  7
USE          B
instruction  8
instruction  9
USE          C
instruction  10
END
```

The location counter order arising from the above sequence is:

```
blank
A
C
B
//
```

and the sequence of the instructions at *load time* will be:

```
instruction  1
             2
             3
             10   (instructions 4 and 5 will
                   be overlaid)
             8
             9
             6
             7
```

The ORG Pseudo-operation

The ORG (Origin) pseudo-operation is used to redefine the value of a location counter. The format of the ORG instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	ORG	Any absolute or positive relocatable expression; or a single virtual symbol (plus or minus a constant, if desired)

The effect of this instruction is to reset the current location counter to the value of the expression in the variable field. The next instruction to be assembled will then be assigned to the new origin.

If there is a symbol in the name field, it is defined as the new origin.

If the variable field is entirely numeric, the ORG is considered absolute, and all symbols defined while this ORG is in effect will be absolute. Thus, the instruction

```
ORG       10000
```

will cause the current location counter to be set to 10000, and the next instruction to be assembled will be assigned to machine location 10000.

On the other hand, to set the location counter to the sixth location of the program, the instruction

```
ORG       START+5
```

must be used, where START is defined as the first location of the program.

If the variable field of an ORG instruction contains a virtual symbol (plus or minus a constant, if desired), then no symbols may be defined under this origin, no literals may be positioned under this origin, and no other explicit or implicit origin derived from any Location Counter pseudo-operation may depend upon this origin.

Data Generation Pseudo-operations

The data generation pseudo-operations are used to enter words of data into the program during assembly. The data might be in the form of octal constants, decimal constants (integers, floating-point numbers, and fixed-point numbers), binary-coded characters, or in a combination of several different forms.

The DUP operation is included in this classification because its most common use is in the generation of tables of data.

The OCT Pseudo-operation

The OCT (Octal Data) pseudo-operation is used to enter binary data expressed in octal form into a program. The format of the OCT instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	OCT	One or more subfields, separated by commas, each containing a signed or unsigned octal integer of n digits, where n $\leqq$ 12

The effect of this operation is to convert each subfield of the variable field to a binary word; these words are assigned to successively higher storage locations as the variable field is processed from left to right. A null subfield causes a word of zeros to be generated; therefore, the number of words of data generated is always one more than the number of commas in the variable field.

The variable field of an OCT instruction may be extended over more than one card, using the ETC instruction; however, a single OCT instruction may not generate more than 60 words.

If there is a symbol in the name field, it is assigned to the first word of data generated.

For example, the instruction

```
ODATA     OCT     777777777777,,77,-66,
```

would generate the following data words:

SYMBOLIC LOCATION	CONTENTS
ODATA	-377777777777
ODATA+1	+000000000000
ODATA+2	+000000000077
ODATA+3	-000000000066
ODATA+4	+000000000000

The DEC Pseudo-operation

The DEC (Decimal Data) pseudo-operation is used to enter binary data expressed in decimal form into a program. The format of the DEC instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	DEC	One or more subfields, separated by commas, each containing a decimal data item (see "Decimal Data Items")

The effect of this operation is to convert each subfield in the variable field into one or two binary words, depending on whether the decimal data item is single- or double-precision. These words are stored in successively higher storage locations as the variable field is processed from left to right. (Note that double-precision numbers will not necessarily be entered into even locations, except where the EVEN pseudo-instruction is used preceding the DEC.) Since a subfield may contain any valid decimal data item, this instruction can be used to generate decimal integers, fixed-point numbers, or floating-point numbers. A null subfield causes a word of zeros to be generated.

The variable field of a DEC instruction may be extended over more than one card, using the ETC instruction; however, a single DEC may not generate more than 60 words.

If there is a symbol in the name field, it is assigned to the first word of data generated.

For example, the instruction

```
DDATA     DEC     13,-22,5B5,1,,
```

would generate the following data words:

SYMBOLIC LOCATION	CONTENTS
DDATA	+000000000015
DDATA+1	-000000000026
DDATA+2	+050000000000
DDATA+3	+000000000001
DDATA+4	+000000000000
DDATA+5	+000000000000

The BCI Pseudo-operation

The BCI (Binary Coded Information) pseudo-operation is used to generate data words consisting of six 6-bit characters in the 7040/7044 standard character code. The format of the BCI instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	BCI	Two subfields, separated by a comma: 1. The count subfield, which consists of a single digit or an immediate symbol that determines the number of *words* to be generated. A null subfield indicates a count of ten. In order to accommodate a full ten words of data on the card, the null subfield must be indicated by a comma in column 12. 2. The data subfield, containing the desired data. The length of this subfield is determined by the count subfield, and it may contain commas, embedded blanks, and/or trailing blanks.

The effect of this operation is to generate, in successively higher storage locations as the data field is processed from left to right, the number of six-character words indicated by the count subfield. Since the count subfield determines the total length of the variable field, the comments field may begin immediately following the data subfield, without the usual blank separator. Similarly, any part of the data subfield that extends beyond the limit indicated by the count subfield will be treated as comments.

If there is a symbol in the name field, it is assigned to the first word of data generated.

For example, the instruction

```
BDATA     BCI     2,BCD MESSAGE
```

would generate the following data words:

SYMBOLIC LOCATION	CONTENTS
BDATA	222324604425
BDATA+1	626221272560

If the count subfield of the instruction above were greater than 2, an appropriate number of data words containing blanks would be generated in locations BDATA+2, etc.

The VFD Pseudo-operation

The VFD (Variable Field Definition) pseudo-operation is used primarily for the generation of data tables. Using this operation, it is possible to prepare packed binary data words containing symbolic, octal, and/or alphameric information. The format of the VFD instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	VFD	Any number of subfields, separated by commas. Each subfield is of one of three types: symbolic, octal, or alphameric; the format of these subfields is detailed below.

If there is a symbol in the name field, it is assigned to the first word of data generated.

Each subfield of the variable field generates zero, one, or more bits of data; thus, the unit of information for this pseudo-operation is the *single bit.*

Each VFD subfield consists of:

1. *The type letter:*
   a. The letter O signifies an octal field.
   b. The letter H signifies an alphameric field.
   c. The absence of either O or H as the first letter signifies a symbolic field.

2. *The Bit Count:* This is given as a decimal integer or as an immediate symbol having an S-value; it specifies how many bits of data are to be generated by the subfield. If an immediate symbol is used, care should be exercised to avoid confusion with a type letter. The immediate symbol will be replaced by its current S-value.

3. *The Separation Character:* slash (/).

4. *The Data Item:* The form of the data item depends on the type of subfield:
   a. In a symbolic subfield, the data consists of one expression.
   b. In an octal subfield, the data consists of one octal integer. The number of digits must be less than or equal to 12.
   c. In an alphameric subfield, the data consists of a string of characters, none of which is a comma or a blank.

Any number of subfields may be used. Successive subfields are converted and packed to the left to form generated data words. If n is the bit count of the first subfield, then the data item in that subfield is converted to an n-bit binary number. This number is placed in the leftmost n bit positions of the first data word to be generated. If n exceeds 36, the leftmost 36 bits of the converted data item form the first generated data word, and the remaining bits are placed in the first (n-36) bit positions of the second generated word. Each succeeding subfield is converted and placed in the leftmost bit positions remaining after the preceding subfield has been processed.

If the total number of bit positions used is not a multiple of 36, then the unused bit positions at the right of the last generated data word will be filled out with zeros.

If the data item is a signed octal integer, the sign is recorded as the high-order bit of the specified bit group.

The data item in a symbolic subfield is evaluated as a symbolic expression and the rightmost 30 bits are retained, complemented if the result was negative. Decimal integers must be less than or equal to 32767.

If the data item is symbolic or octal and occupies more than n bits, only the rightmost n bits of the converted data item are used. If the data item occupies fewer than n bits, sufficient zero bits are placed at the left of the converted data item to form an n-bit binary number. Neither condition is regarded as an error by the assembler.

The data item in an alphameric subfield may consist of any combination of characters other than a comma or a blank. Each character is converted to its 6-bit binary equivalent. If the data item occupies more than n bits, only the rightmost n bits are used. If it occupies fewer than n bits, sufficient 6-bit groups of the form 110000 (the internal code for blank) are placed at the left of the converted data item to form an n-bit binary number; if n is not a multiple of 6, the leftmost character used, or the leftmost blank, is truncated. None of these conditions is regarded as an error by the assembler.

For example, suppose the programmer would like to break up a single 36-bit word as follows:

BIT POSITIONS	CONTENTS
S, 1-9	Binary equivalent of the decimal integer 895
10-14	Binary equivalent of the octal integer 37
15-20	Binary code for the character C
21-35	Binary value of the symbol ALPHA

The instruetion to generatc this word is

```
VFD     10/895,O5/37,H6/C,
          15/ALPHA
```

The DUP Pseudo-operation

The DUP (Duplieate) pseudo-operation eauscs a card or sequcnce of eards to be duplieated a speeified number of times. The format of the DUP instruetion is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	DUP	Two subfields, separated by a eomma: 1. The eard eount 2. The iteration eount Each subfield eonsists of one symbolic expression containing integers and/or immediate symbols. Immediate symbols will be replaced by their eurrent S-values.

Let m stand for the eard eount and n for the iteration eount. Then the meaning of the instruetion

```
DUP     m, n
```

is "duplieate the binary data gencrated by the next m card images n times."

The m eards following the DUP are referred to as the *range* of the DUP. If the iteration eount is zero, the cards within the range of the DUP will be ignored. If the iteration count is one, the DUP statement is ignored.

For example, the sequenee

```
DUP     2, 3
PZE     X
PZE     Y
```

results in the sequence

```
PZE     X
PZE     Y
PZE     X
PZE     Y
PZE     X
PZE     Y
```

If there is a symbol in the name field of a DUP, it refers to the first word generated by instruetions within the range of the DUP.

If a statement within the range of thc DUP has a symbol in the name field, then this symbol appears only in the first iteration of the rangc except in the following eases when the name field appears in every iteration:

1. Operation requires a name field (i.e., BOOL, EQU, SYN, MACRO, MAX, MIN, OPD, OPSYN, SAVE, or SET operations).
2. Opcration is REM, TTL, or CONTRL.
3. The name field contains an asterisk in column 1 (i.e., a comments card).

A DUP will not take effect if an END card is encountered within thc range of thc DUP.

A DUP may not be used within the range of another DUP.

Whcn a DUP is generated by a macro-operation or when a maero-operation falls within the range of a DUP, eertain eard imagcs may not be ineluded within the eount. See the seetion "DUP With Maero-operations" for a detailed explanation.

Storage Allocation Pseudo-operations

The BSS and BES pseudo-operations are used to reserve eorc storage areas as data storage areas or work areas. The EVEN pseudo-operation causes the next word gencrated to be assigned to an even storage address, and is used to ensure that double-precision numbers are properly cntered into core storage.

The BSS Pseudo-operation

The BSS (Bloek Started by Symbol) pseudo-operation is used to reserve an area of core storage within a program for data storage or working spaee. The format of this instruetion is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	BSS	Any absolute expression

BSS performs two functions:

1. A bloek of conseeutive storage loeations is reserved. The number of locations reserved is the value of the expression in the variable field; the location of thc block is detcrmined by the value of the eurrent location counter when the BSS is eneountered.
2. If there is a symbol in the name field, it is assigned to the first loeation of the bloek reserved by the BSS.

BSS eauses an area to be skipped, not cleared, and therefore it may not be assumed that an area reserved by a BSS contains zeros.

Consider the following example:

```
ALPIIA      IORD      BETA,,4
BETA        BSS       4
GAMMA       IORD      DELTA,,6
```

Assumc that the symbol ALPHA has been assigned to location 1001. Thcn thc symbol BETA will be assigned to loeation 1002, and thc symbol GAMMA to loeation 1006, leaving four locations (1002-1005) for the bloek BETA.

A deseription of the extended variable field speeifying mode and dimensions for load-time debugging ean bc found in the seetion "Supplying Modal Information to the Debugging Dictionary" of the manual *IBM 7040/7044 Operating System (16/32K): Debugging Facilities*, Form C28-6803.

The BES Pseudo-operation

The BES (Block Ended by Symbol) pseudo-operation is used to reserve an area of core storage within a program for data storage or working space. The format of the BES instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	BES	Any absolute expression

BES performs two functions:

1. A block of consecutive storage locations is reserved. The number of locations reserved is the value of the expression in the variable field; the location of the block is determined by the value of the current location counter when the BES is encountered.
2. If there is a symbol in the name field, it is assigned to the next location following the location of the block.

BES causes an area to be skipped, not cleared, and therefore it may not be assumed that an area reserved by a BES contains zeros.

Consider the following example:

```
ALPHA     IORD    BETA,,4
BETA      BES     4
GAMMA     IORD    DELTA,4
```

Assume that the symbol ALPHA has been assigned to location 1001. Then both the symbol BETA and the symbol GAMMA will be assigned to location 1006, leaving four locations (1002-1005) for the reserved block BETA.

Note that if the name fields are left blank, BES and BSS have the same effect, and that the three sequences

```
1. ALPHA     BES     25
             CLA     BETA
2.           BES     25
   ALPHA     CLA     BETA
3.           BSS     25
   ALPHA     CLA     BETA
```

are effectively the same.

A description of the extended variable field specifying mode and dimensions for load-time debugging can be found in the section "Supplying Modal Information to the Debugging Dictionary" of the manual *IBM 7040/7044 Operating System (16/32K): Debugging Facilities,* Form C28-6803.

The EVEN Pseudo-operation

The EVEN pseudo-operation causes the next word generated to be assigned to an even address and is used to ensure an even load address for the data or instruction that follows – usually, a double-precision floating-point number. The format of the EVEN instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	EVEN	Blanks

The effect of this operation is to force the load address of the next word to an even number. If the load address is odd when an EVEN operation is given, an AXT 0, 0 instruction is generated in the odd location and the next word is assigned to the next (even) location. In an ABSMOD assembly, this function is performed by the assembler at assembly time. In a RELMOD assembly, it is performed by the Loader at load time.

Symbol Definition Pseudo-operations

Except for a few of the pseudo-operations, any operation may be used to define a symbol simply by placing the symbol to be defined in the name field of that operation. However, the symbol definition pseudo-operations (EQU, SYN, MAX, MIN, SET, and BOOL) exist solely for the purpose of defining symbols.

The EQU and SYN Pseudo-operations

The EQU (Equals) and SYN (Synonym) pseudo-operations are used to assign to a symbol a value other than that of the value of the location counter in control when the symbol is encountered. The format of the EQU and SYN instructions is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	EQU or SYN	An absolute or relocatable expression, or a single virtual symbol (plus or minus a constant, if desired)

The effect of the EQU (SYN) operation is to define the symbol in the name field as having the value of the expression in the variable field (and the same relocation properties).

If an asterisk (*) is used as an element in the variable field of an EQU (SYN), the value assigned to the asterisk is the next location to be assigned by the assembler. For example, in the sequence

```
          CLA     TMP1
FSTL      EQU     *
          ADD     TMP2
```

if the CLA instruction is assigned to location 0102, the symbol FSTL would then be defined as having the value 0103, and the ADD instruction would be assigned to location 0103.

A description of the extended variable field specifying mode and dimensions for load-time debugging can be found in the section "Supplying Modal Information to the Debugging Dictionary" of the manual *IBM 7040/7044 Operating System (16/32K): Debugging Facilities,* Form C28-6803.

The MAX Pseudo-operation

The MAX (Maximum) pseudo-operation defines a symbol as having a value equal to that of the expression in the variable field having the greatest value. The format of the MAX instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	MAX	A series of non-complex expressions, separated by commas

The MAX operation acts as an EQU, using the value and relocation properties of the expression in the variable field that has the greatest value. If two or more expressions are equal in value, the first will be used.

In a relocatable assembly, comparisons are made using the value assigned at assembly time. For example, in the sequence

```
ALPHA     PZE     X
            .
            .
            .
BETA      PZE     Y
            .
            .
            .
DELTA     MAX     ALPHA+20, BETA
```

DELTA will be defined as ALPHA+20 if the assembler generates 20 words or less between ALPHA and BETA; it will be defined as BETA if the assembler generates more than 20 words between ALPHA and BETA.

The MIN Pseudo-operation

The MIN (Minimum) pseudo-operation defines a symbol as having a value equal to that of the expression in the variable field having the least value. The format of the MIN instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	MIN	A series of non-complex expressions, separated by commas

MIN is the opposite of MAX; it acts as an EQU, using the value and relocation properties of the expression in the variable field that has the least value.

The SET Pseudo-operation

The SET pseudo-operation is used to define immediate symbols for use in instructions. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	SET	Any symbolic expression

The effect of this operation is to assign the numeric value of the expression in the variable field to the symbol appearing in the name field, regardless of any prior immediate value of the symbol. This value is then referred to as the "current S-value" of the symbol.

The expression in the variable field is evaluated using the current S-values of any symbols appearing in it. It is evaluated as a 35-bit signed integer and is then truncated to 15 bits. If it is negative, it is complemented prior to truncation.

If a symbol appearing in the variable field has not been defined as an immediate symbol by a SET instruction, it has an S-value of 1 if it has already been defined as an ordinary symbol, or 0 if it has not. This is particularly useful in the IFF and IFT pseudo-instructions in determining whether a symbol has already been defined.

An immediate symbol may be redefined as many times as desired by subsequent SET instructions, but no immediate symbol may duplicate an ordinary symbol in the program.

The BOOL Pseudo-operation

The BOOL (Boolean Constant) pseudo-operation is used to define a symbol as an octal constant. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	BOOL	An unsigned octal integer of five digits or less

The effect of this operation is to define the symbol in the name field as having the value of the octal constant in the variable field.

Program Section Pseudo-operations

The program section pseudo-operations (QUAL and ENDQ) provide a means of dividing a program into sections by "qualifying" all of the symbols defined in a given section with an additional symbol. Other sections of the program can then refer to a symbol within a qualification section by using that symbol in combination with the qualification symbol. This is particularly useful when parts of the same program are written at different times or by different programmers, in that it eliminates the possibility of inadvertent duplication of symbols from one part to the next.

The QUAL Pseudo-operation

The QUAL (Qualify) pseudo-operation is used to indicate the beginning of a qualification section and its associated qualification symbol. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	QUAL	A single symbol

The symbol in the variable field becomes a qualifier for all symbols defined within the range of the section controlled by the QUAL operation (ENDQ is used to delimit the range; see below). References to a name defined in a qualification section from within the same section need not be qualified. References from outside the section are qualified by placing the qualifier in front of the desired symbol, with a connecting dollar sign ($). For example, to refer to a symbol ALPHA that is defined in qualification section QS1, the symbol is

written QS1$ALPHA. To refer to a symbol BETA in a section that is not qualified from within a qualification section, the notation $BETA is used. The nonqualified section may be considered as effectively having a blank qualifier.

Qualification sections may be nested to provide multiple qualification. The range (from a QUAL to its corresponding ENDQ) of a lower-level QUAL must fall completely within the range of the next higher-level QUAL. A symbol is automatically qualified by any qualifiers of a higher level than the highest one specified in using the symbol. A multiply-qualified symbol can be referred to without using all of its qualifiers, provided enough qualifiers are given to uniquely determine the symbol. In any case, the qualifiers must be specified in the same order that the nesting occurs. Below are several examples illustrating qualification.

In the sequence

```
        QUAL    H
A       BSS     1
        CLA     X
        ENDQ    H
        QUAL    J
A       BSS     1
        ENDQ    J
```

(Qualification section H; Qualification section J)

if X is written as A or H$A, it refers to the first definition of A;

if X is written as J$A, it refers to the second definition of A.

In the sequence with nested qualification

```
        QUAL    M
A       BSS     1
        QUAL    N
        CLA     X
        ENDQ    N
        ENDQ    M
A       BSS     1
        CLA     Y
```

(Qualification section M$N; Qualification section M)

if X is written as A, it refers to the first definition of A; if it is written as $A, it refers to the second (nonqualified) A.

if Y is written as A, it refers to the second A; if it is written as M$A, it refers to the first.

In the more complicated sequence

```
        QUAL    ONE
A       BSS     1
        QUAL    TWO
A       BSS     1
        CLA     X
        ENDQ    TWO
        CLA     Y
        ENDQ    ONE
        QUAL    THREE
A       BSS     1
        QUAL    TWO
A       BSS     1
        ENDQ    TWO
        ENDQ    THREE
```

(Qualification section ONE$TWO; Qualification section ONE; Qualification section THREE$TWO; Qualification section THREE)

the references taken for X and Y are as follows:

if X is written as ONE$A, it refers to the first definition of A; if it is written as A, TWO$A, or ONE$TWO$A, it refers to the second A; if it is written as THREE$A, it refers to the third A; if it is written as THREETWOA, it refers to the fourth A.

if Y is written as A or ONE$A, it refers to the first definition of A; if it is written as TWO$A or ONE$TWO$A, it refers to the second A; if it is written as THREE$A, it refers to the third A; if it is written as THREETWOA, it refers to the fourth A.

It should be noted that the two sections TWO are distinct, and not separate parts of the same section. The first is section ONE$TWO and the second, section THREE$TWO.

The ENDQ Pseudo-operation

The ENDQ (End Qualification) pseudo-operation terminates the range of a qualification section. The format of the ENDQ instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ENDQ	A symbol or blanks

ENDQ terminates the qualification section whose qualifier appears in the variable field. If the variable field is blank, the innermost qualification section is terminated.

In the case of nested qualification, an ENDQ with a higher-level qualifier in the variable field will also terminate all lower-level qualification sections. For example, in the sequence

```
        QUAL    ALPHA
          .
          .
          .
        QUAL    BETA
          .
          .
          .
        QUAL    GAMMA
          .
          .
          .
        ENDQ    ALPHA
```

the last instruction terminates the qualification sections ALPHA$BETA and ALPHA$BETA$GAMMA, as well as the section ALPHA.

Literal Positioning Pseudo-operations

Two pseudo-operations (LORG and LITORG) are provided for controlling the location at which the literals used within a program are to be placed.

The LORG Pseudo-operation

The LORG (Literal Pool Origin) pseudo-operation controls the positioning of all of the literals used in a pro-

gram except those positioned under LITORG control. The format of the LORG instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	LORG	Blanks

The effect of this operation is to position the Literal Pool, starting at the location of the LORG in the symbolic deck. If LORG is not specified, the Literal Pool origin is the final value of the location counter in use at the end of the program, plus one. There may be only one LORG in a program.

The LITORG Pseudo-operation

The LITORG (Literal Origin) pseudo-operation enables literals used within a block of coding to be associated with that block. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	LITORG	Blanks

The effect of this operation is to position all of the literals encountered *up to the* LITORG (either from the beginning of the program or from a previous LITORG), starting at the location of the LITORG in the symbolic deck. Any literals encountered following a LITORG will be positioned at the next LITORG, or, if there is none, they will be positioned under the control of LORG.

Note that duplicate literals under control of a LORG will be eliminated; however, under LITORG control, duplicate literals will be eliminated only within each LITORG section.

The primary uses of LITORG are for associating literals used within a control section (see "Control Dictionary Pseudo-operations") with that control section, and in the assembly of multiphase programs.

Conditional Assembly Pseudo-operations

The conditional assembly pseudo-operations (IFT and IFF) are used to specify that an instruction is to be assembled only when certain criteria are satisfied.

The IFT and IFF Pseudo-operations

The IFT (If True) and IFF (If False) pseudo-operations allow conditional assembly of the following instruction. The format of these instructions is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	IFT(IFF)	One subfield, consisting of two symbolic expressions, separated by an equal sign (=); or a single expression.

The effect of IFT is to cause the following instruction to be assembled only if the two symbolic expressions separated by the equal sign have the same value. The effect of IFF is just the opposite; the following instruction will be assembled only if the two expressions are unequal in value.

If only one expression is present and there is no equal sign, then =0 will be assumed.

The expressions in the variable field are evaluated using the current S-values of any symbols which are present.

For example, the sequence

```
A       SET     11
        IFT     A+25=B*3
        CLA     X
        IFT     A+25=B*3+3
        CLA     Y
        STO     Z
```

would produce

```
        CLA     X
        STO     Z
```

if B had been previously defined by

```
B       SET     12
```

If, on the other hand, B had been previously defined by

```
B       SET     11
```

the sequence produced would be

```
        CLA     Y
        STO     Z
```

Both IFT and IFF may be used anywhere in a program, and may precede any machine instruction, pseudo-instruction, or macro-instruction.

Operation Definition Pseudo-operations

The operation definition pseudo-operations (OPD and OPSYN) are provided to allow the programmer to define his own operation codes.

The OPD Pseudo-operation

The OPD (Operation Definition) pseudo-operation defines a symbol as a machine operation code. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	OPD	Five subfields, separated by commas, as follows: 1. An octal machine operation code skeleton; i.e., a 12-digit signed or unsigned octal integer. The digit representing the tag field must be zero. 2. An asterisk if indirect addressing is permissible; otherwise, this subfield must be null. 3. The minimum number of subfields permitted in this operation. 4. The maximum number of subfields permitted in this operation. 5. The number of bits allowed in the decrements (0-6 or 15). All of the subfields must be present and in the order indicated.

The effect of this operation is to define the symbol in the name field as the operation code for the operation definition in the variable field.

For example, to define an operation CAD to be the same as the machine operation CLA, the following instruction would be used:

```
CAD         OPD         050000000000,*,1,2,0
```

If the symbol in the name field is the same as an existing machine operation code, that code is defined and the new definition replaces the former. For example, if a programmer desired to allow three subfields in a TSX instruction where the third subfield might be a subroutine parameter, he would write

```
TSX         OPD         007400000000,,2,3,6
```

The OPSYN Pseudo-operation

The OPSYN (Operation Synonym) pseudo-operation is used to define synonyms for existing operation codes. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	OPSYN	An operation code

The effect of this operation is to define the symbol in the name field as the equivalent of the operation code in the variable field. Thereafter, the two codes may be used interchangeably.

The operation code ***, ENDM, ETC, IRP, or MACRO may not be used in either the name field or the variable field of an OPSYN instruction. If the operation code BCI, REM, or TTL appears in an OPSYN instruction, neither it nor its equivalent may appear in a macro-definition or macro-expansion.

Subroutine Pseudo-operations

The subroutine pseudo-operations (CALL, SAVE, and RETURN) are provided to aid the programmer in preparing closed subroutines and the linkages to them from the main program.

The CALL Pseudo-operation

The CALL psuedo-operation is used to produce a standard subroutine calling sequence. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	CALL	One or more subfields, as follows: 1. A symbol (the name of a subroutine), or **. 2. The arguments of the calling sequence (if any) enclosed within parentheses and separated by commas. These may be any symbolic expression. 3. The error returns (if any), separated by commas. 4. An identification symbol (if desired) of up to six alphameric characters, delimited by apostrophes.

The general form of a CALL instruction is:

```
symbol      CALL        name(arg1, arg2, . . . , argn)
                            ret1, ret2, . . . , retm'id'
```

where name is the name of a subroutine, arg1, arg2, . . . , argn are the arguments of the calling sequence; ret1, ret2, . . . , retm are the error returns, and 'id' is the identification symbol. If 'id' is specified, this symbol will appear in the calling sequence if it is not specified, the symbol in the name field will appear in the calling sequence instead.

Note that no comma precedes the left parenthesis or follows the right parenthesis or precedes the 'id'.

If the subroutine name is not included in the same program, then the programmer must identify the name as external through an EXTERN instruction (see below).

Expansions of the CALL Instruction: The following are examples of the calling sequences generated by various CALL instructions. ARG is used to denote an argument and RET to denote a return.

```
INSTRUCTION                                  EXPANSION
LCS CALL NAME(ARG1, . . . , ARGn)RETN, . . . ,
RETm'IDENT'
                                     LCS TSL   NAME
                                         TXI   *+2+n+m,,n
                                         BCI   1,IDENT
                                         PZE   ARG1
                                          .
                                          .
                                          .
```

INSTRUCTION		EXPANSION	
		PZE	ARGn
		TRA	RETm
		.	
		.	
		.	
		TRA	RET1
LCS CALL NAME(ARG1, ARG2)			
	LCS	TSL	NAME
		TXI	*+2+2+0,,2
		BCI	1,LCS
		PZE	ARG1
		PZE	ARG2
LCS CALL NAME,RET (see note following)	LCS	TSL	NAME
		TXI	*+2+0+1,,0
		BCI	1,LCS
		TRA	RET
LCS CALL NAME'ID' (see note following)	LCS	TSL	NAME
		TXI	*+2+0+0,,0
		BCI	1,ID

Note: The first subfield of the variable field (i.e., the name of the subroutine) may be delimited by a comma, a left parenthesis, an apostrophe, or a blank.

The SAVE Pseudo-operation

The SAVE pseudo-operation is used to produce the instructions necessary for the saving and restoring of the index registers used by a program, and for the updating of s.SLOC. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	SAVE	Up to three numeric subfields, separated by commas, specifying the particular index registers to be restored. Any or all of them may be specified, and in any order.

Index register 4 is always automatically saved and restored, but may still be specified.

Index registers 1 and 2 are always automatically saved but are restored upon exit from the subprogram only if they are specified in the variable field of the SAVE instruction. If 1 and/or 2 are not specified, the second and/or third words, respectively, of the expansion (see below) will have tag fields of zero.

s.SLOC, which is automatically updated by the SAVE operation, is a standard communication location.

If the name associated with the SAVE instruction is to be referred to from outside the program in which it appears, it must also appear in the variable field of an ENTRY instruction.

Expansion of the SAVE Instruction: The following is an example of the instructions generated by the SAVE instruction.

INSTRUCTION		EXPANSION	
NAME SAVE 1,2,4			
		BCI	1, NAME
		AXT	**, 1
		AXT	**, 2
		AXT	**, 4
		SXA	S. SLOC, 4
		AXT	**, 4
	NAME	TXI	**
		SXA	*−2, 4
		LXA	S. SLOC, 4
		SXA	*−6, 4
		LAC	NAME, 4
		TXI	*+1, 4, 1
		SXA	S. SLOC, 4
		SXA	*−11, 2
		SXA	*−13, 1

The RETURN Pseudo-operation

The RETURN pseudo-operation is designed to be used with CALL and SAVE, making use of the error (or alternate) returns option provided with these operations. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	RETURN	One subfield, or two subfields, separated by a comma, as follows: 1. The name of the associated SAVE instruction. (This subfield must be present.) 2. The number of the alternate (or error) return desired. This may be an integer, an immediate symbol, or an absolute expression.

An error or alternate return should be specified in the second subfield only if the associated CALL statement provided for returns.

Expansions of the RETURN Instruction: The following are examples of the instructions generated by the RETURN instruction.

INSTRUCTION		EXPANSION	
To specify the normal return:			
LCS RETURN NAME			
	LCS	BSS	0
		TRA	NAME−5
To specify the *ith* return (where zero is the normal return):			
LCS RETURN NAME,i			
	LCS	BSS	0
		LXA	NAME, 4
		SXA	*+1, 4
		LXA	**, 4
		TIX	*+1, 4, i
		SXA	NAME, 4
		TRA	NAME−5

Control Dictionary Pseudo-operations

An IBMAP assembly always produces a Control Dictionary, which is the first part of the binary object program deck. One of the primary purposes of this

dictionary is to provide the information necessary to enable the Loader to make references between program segments that are assembled separately but are loaded together and refer to each other. The Control Dictionary pseudo-operations are the means by which the user makes entries into this dictionary.

Each Control Dictionary entry made by the user consists of two words: the first word contains a name that provides external identification for this entry; the second word gives the length of the entry and, in the case of control sections and entry points, its position in the deck relative to the beginning of the program segment.

There are four types of entries that can be made by the user: control section names, file names, entry point symbols, and external symbols. The length of a control section must be nonzero; entries for files, entry points, and external symbols have zero length.

Control Section Names: These entries are made using the CONTRL pseudo-operation. The mode of the limits of a control section is assumed to be the same as the mode of the assembly; therefore, in a RELMOD assembly, absolute symbols may not be specified as the limits of a control section. Control sections may not be overlapped or nested.

File Names: These entries are made using the FILE pseudo-operation. The file name is entered into the dictionary and refers to the file control block associated with that file.

Entry Point Symbols: These entries are made using the ENTRY pseudo-operation and define the points at which other programs can refer to this assembly. The mode of the entry points is assumed to be the same as the mode of the assembly; therefore, in a RELMOD assembly, absolute symbols may not be specified as entry points.

External Symbols: These entries are made using the EXTERN pseudo-operation. Symbols that are identified as external must be virtual in the assembly; i.e., they may not be defined in this assembly. They will be defined by the Loader at load time.

When the separate assemblies are loaded together, the several Control Dictionaries are examined by the Loader to determine the correct values necessary to transfer control between program segments, to obtain data from other program segments, etc.

The CONTRL Pseudo-operation

The CONTRL (Control Section) pseudo-operation is used to define a control section and to make the proper entry for it into the Control Dictionary. The format of the CONTRL instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Control section name (other than //) or blanks	CONTRL	One of the following: 1. Two subfields, each an expression, separated by a comma 2. A single location counter name 3. A single qualification symbol

The contents of the name field are entered into the Control Dictionary as the name of the control section. If the name field is blank, then the first name in the variable field will be used. The length and initial location of the section is dependent on the content of the variable field, as follows:

1. If the variable field contains two subfields, each an expression, the length of the control section is the difference of the values of the second and first subfields, and the initial location is the value of the first subfield.

2. If the variable field contains a location counter name or a qualification symbol, the length of the control section is the same as the range of the location counter or qualification section named, and the initial location is the first instruction within the range.

The length of a section is determined by its first and last locations, rather than by the number of locations coded between these points; therefore, the user is cautioned to consider the effect of any ORG pseudo-instructions within a control section.

Symbols whose locations fall within the limits of a control section will be treated as part of that control section. A symbol defined as relative to a location within a control section will not be considered as part of the control section if its location falls outside the limits of the control section.

If it is desired to obtain the blank COMMON area of 7040/7044 FORTRAN IV, then only the BSS and BES pseudo-instructions may appear under the // counter; the last location counter in location counter sequence should end at the highest location in the program (exclusive of blank COMMON); and a control section of the form

```
CONTRL  //
```

must be given. Note that the name field must be blank.

For example, in the sequence

```
CONTRL   //
USE      A
  .
  .
  .
USE      B
  .
  .
  .
USE      //
BSS      20
USE      C
  .
  .
  .
USE      A
  .
  .
  .
END
```

the blank COMMON counter, //, will have its initial location defined as one higher than the last value reached by location counter C.

The ENTRY Pseudo-operation

The ENTRY pseudo-operation identifies a symbol as an entry point in the program and makes an entry in the Control Dictionary. The format of the ENTRY instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ENTRY	A symbol, which may be qualified

The symbol in the variable field must be an ordinary symbol, defined elsewhere in the program. It is entered into the Control Dictionary as an entry point. The symbol may be qualified, in which case the base (rightmost) symbol is used. The symbol should not be defined under the // (blank COMMON) control section.

The EXTERN Pseudo-operation

The EXTERN (External Symbol) pseudo-operation is used to identify a symbol as external and to make the appropriate entry into the Control Dictionary. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	EXTERN	One or more subfields, separated by commas, each containing a single virtual symbol

This operation identifies each symbol in the variable field as an external symbol for purposes of reference throughout the assembly. Each symbol causes a separate entry of zero length in the Control Dictionary.

EXTERN instructions will be generated automatically for any symbols of the form S.xxxx used within a program but not defined in the name field of an instruction in that program.

Any virtual symbol in a program that is not identified by an EXTERN instruction will be treated as undefined.

File Description Pseudo-operations

The file description pseudo-operations, FILE and LABEL, are used to specify the requirements of any input/output files used by a program and are very similar to the Loader control cards $FILE and $LABEL.

All FILE and LABEL pseudo-instructions must precede any instructions in the program except for comments cards of the type with an asterisk in column 1 and list control pseudo-operations.

Notation Conventions: In the discussion of FILE, the following notation conventions are used:

1. Material in brackets [] indicates that the enclosed material may be omitted, in which case the underscored option, if any, will be used.
2. Material in braces {} indicates that a choice of the contents is to be made by the user.
3. Upper-case words, if used, must be present in the form indicated.
4. Lower-case words represent generic quantities whose values must be supplied by the user.

The FILE Pseudo-operation

The FILE pseudo-operation is used to describe the characteristics of input/output files used by the program. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol	FILE	From 2 to 21 subfields, separated by commas, as listed below. The first two subfields must be present and must contain the unit assignments; the order of the remaining subfields is not important, and, if one is omitted, the standard case will be assumed.

The FILE pseudo-operation causes the symbol in the name field to be entered into the Control Dictionary as the external name of the file, enabling other assemblies to refer to this file description (file control block).

The subfields of the variable field are discussed in detail in the $FILE card section of the 7040/7044 Programmer's Guide publication and, therefore, are only listed here:

1. Unit Assignment Option

Unit-1, unit-2

2. Mounting Option

[,{ MOUNT / READY / DEFER }] and/or [,{ MOUNTi / READYi / DEFERi }]

3. File Usage Option

[,CKFILE]

4. Block Size Specification

, BLOCK = xxxx

where xxxx is a number that specifies the block size for this file.

5. Buffer Options

[,{ SINGLE / DOUBLE }]

6. Reel Handling Options

[,{ REEL / REELS }]

7. File Density Options

[,{ HIGH / LOW }]

8. Mode Option

[,MIXED]

9. Block Sequence Options

[,{ SEQ / NOSEQ }]

10. Check Sum Options

[,{ CKSM / NOCKSM }]

11. Checkpoint Options

[,{ CKAFLB / CKCKFL / CKLBFL / NOCKPT }]

12. File Close Options

[,{ PRINT / PUNCH / HOLD / SCRTCH }]

13. Labeling Options

[,{ ADDLBL=symbol / NSLBL=symbol }]

where symbol is the name of a routine that processes additional label fields (ADDLBL) or nonstandard labels (NSLBL). This symbol must be defined in this assembly by an EXTERN or ENTRY operation or as a control section name.

14. Record Format Options

[,{ TYPE1 / TYPE2 / TYPE3 }]

15. Record Length Option

[,LRL=xxx]

where xxx is a number that specifies the length of a logical record in this file.

16. Record Count Option

[,RCT=xxx]

where xxx is the number of logical records in a block.

17. Error Option

[,ERR=symbol]

where symbol is the name of a routine to be entered on an error condition. This symbol must be defined in this assembly by an EXTERN or ENTRY operation or as a control section name.

18. End of Reel Option

[,EOR=symbol]

where symbol is the name of a routine to be entered when an end of reel is encountered. This symbol must be defined in this assembly by an EXTERN or ENTRY operation or as a control section name.

19. End of File Option

[,EOF=symbol]

where symbol is the name of a routine to be entered when an end of file is encountered. This symbol must be defined in this assembly by an EXTERN or ENTRY operation or as a control section name.

The LABEL Pseudo-operation

The LABEL pseudo-operation is used to further describe a file. This instruction, which must immediately follow the FILE instruction that describes the file to be labeled, is of the format:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	LABEL	Up to four subfields, separated by commas, as follows: 1. File serial number: up to five alphameric characters 2. Reel sequence number: up to four decimal digits 3. Date or days: up to four decimal digits; or two decimal digits followed by a slash, followed by up to three decimal digits 4. Identification: a string of ten characters

The reader is referred to the $LABEL card section of the 7040/7044 Programmer's Guide publication for the details of the parameters of this card.

List Control Pseudo-operations

The list control pseudo-operations provide the user with the ability to control the form of the assembly output listing. More specifically, these pseudo-operations are used to indicate what is to be listed, to indicate spacing and page ejection, to print subtitles for pages, and to prepare an index of important locations within an assembly.

The PCC Pseudo-operation

The PCC (Print Control Cards) pseudo-operation is used to cause the listing of list control pseudo-instructions that would not otherwise be listed. The format of the PCC instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	PCC	One of: ON, OFF, or blanks

PCC controls listing of the following list control pseudo-instructions: DETAIL, EJECT, INDEX, LIST, PMC,

SPACE, TITLE, TTL. (The PCC instruction itself will always be listed.)

PCC ON causes the listing of these cards; PCC OFF suppresses listing of these cards and is the normal mode. If the variable field is blank, the current setting of PCC will be inverted.

The SPACE Pseudo-operation

The SPACE pseudo-operation is used to generate one or more blank lines in the assembly listing. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	SPACE	An expression

The expression in the variable field is evaluated, using the current S-values of any symbols in the expression. This value is the number of blank lines that will appear in the listing, except that, if the value is zero, one blank line will appear.

The SPACE instruction is not itself listed except where the mode of PCC is ON.

The EJECT Pseudo-operation

The EJECT pseudo-operation causes the next line of the listing to appear at the top of a new page. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	EJECT	Any information

EJECT is not itself listed except where the mode of PCC is ON.

The TITLE Pseudo-operation

Most symbolic instructions generate one binary word in the object program; some pseudo-operations generate no binary words; and some pseudo-operations may generate several binary words. The pseudo-operations in the last category (BCI, DEC, DUP, OCT, and VFD) are called *generative pseudo-operations*. Normally, the assembly listing will contain all of the binary words generated by these pseudo-operations. The TITLE pseudo-operation is used to abbreviate the assembly listing by eliminating from the listing all but the first word generated by any of the generative pseudo-operations. (In the case of DUP, the original sequence will appear, but iterations are eliminated from the listing.) The format of the TITLE instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	TITLE	Any information

Following the occurrence of a TITLE, and until the next occurrence of a DETAIL instruction (see below), the assembler will exclude from the listing any line that contains octal information but not the symbolic instruction which caused it to be generated.

TITLE will not itself be listed except where the mode of PCC is ON.

The DETAIL Pseudo-operation

The DETAIL pseudo-operation is used to resume the listing of generated data after such listing has been suspended by a TITLE instruction. The format of the DETAIL instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	DETAIL	Any information

The DETAIL instruction will not itself be listed except where the mode of PCC is ON.

The PMC Pseudo-operation

In its normal mode of operation, the assembler does not list the card images generated by a macro-instruction or the subroutine pseudo-instructions. The PMC (Print Macro Cards) pseudo-operation is used to cause these cards to be listed. The format of the PMC instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	PMC	One of: ON, OFF, or blanks

PMC ON causes listing of the card images generated by macro-instructions, CALL, SAVE, and RETURN. PMC OFF suppresses such listing and is the normal mode. A blank variable field inverts the current setting of PMC.

PMC will not itself be listed except where the mode of PCC is ON.

The TTL Pseudo-operation

The TTL (Subtitle) pseudo-operation is used to place a subheading on each page of the listing and to initiate renumbering of the pages in the listing. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks or a decimal integer	TTL	A string of characters (the desired subtitle), starting in column 13. The string may contain any MAP character, and may also include embedded blanks.

The appearance of a TTL causes a page ejection.

The effect of the TTL operation is to generate a subheading on each page of the listing. Card columns 13-72 are printed in words 4-13 of a subheading line for each page until changed by another TTL or until deleted by a TTL with a blank variable field.

A decimal integer (from 1 to 32768) in the name field will cause a renumbering of pages, beginning with that number. If the name field is blank, the page numbering sequence will not be changed.

The INDEX Pseudo-operation

The INDEX pseudo-operation is used to generate in the assembly listing an index of specified symbols and the definitions assigned to them. The format of the INDEX instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	INDEX	A series of symbols, separated by commas

The INDEX operation generates a table of contents of any desired locations in the program; i.e., it lists the symbols specified in the variable field along with the definition assigned to each. The first appearance of an INDEX instruction will cause the heading

TABLE OF CONTENTS

NAME VALUE CONTROL

to be printed preceding the index. Subsequent INDEX instructions will not cause this heading to be printed.

The UNLIST Pseudo-operation

The UNLIST pseudo-operation causes all listing to be suspended. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	UNLIST	Any information

The UNLIST instruction is itself listed (unless a previous UNLIST is still in effect), but thereafter no lines will be listed until a LIST pseudo-instruction (see below) is encountered.

Note that, even though no actual listing takes place, the list control pseudo-instructions DETAIL, EJECT, PCC, TITLE, and TTL will still be effective under UNLIST control. For example, if the mode of PCC is OFF when the UNLIST is encountered and a subsequent PCC ON is used, the mode of PCC will be ON when listing is resumed. However, only one page ejection will occur, regardless of the number of EJECT or TTL pseudo-instructions encountered under UNLIST control.

The LIST Pseudo-operation

The LIST pseudo-operation is used to resume the assembly output listing following an UNLIST. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	LIST	Any information

The LIST instruction will not itself appear in the assembly listing except where the current mode of PCC is ON, but it will cause one blank line to appear in the listing whether or not an UNLIST is in effect.

Miscellaneous Pseudo-operations

Of the seven pseudo-operations in this category, two (ABS and FUL) are ABSMOD assembly pseudo-operations (i.e., they are only effective in an ABSMOD assembly); one (REM) is a remarks card; one (ETC) is used to extend the variable field of an instruction over more than one card; one (NULL) is effectively a "no operation"; one (END) is used to indicate the end of the symbolic deck; and one (TCD) is used to indicate the end of a memory load.

The FUL Pseudo-operation

The FUL (Full) pseudo-operation is used to specify card output in the 24-words-per-card "full" mode. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	FUL	Blanks

Column-binary cards or card images produced in the full mode contain the first word of output in columns 1-3, the second word in columns 4-6, and so on, to a maximum of 24 words per card.

Regardless of whether the full mode is already in force, the effect of a FUL instruction on the binary output is to cause any words remaining in the punch buffer to be written out and the next output to start at the beginning of a new card. Binary output will thereafter be in the full mode until the end of the assembly or until an ABS instruction (see below) is encountered.

FUL is effective only in an ABSMOD assembly; in a RELMOD assembly, it will be ignored.

The ABS Pseudo-operation

The ABS (Absolute Punch Mode) pseudo-operation is used to return from the full mode of assembly to the normal punching mode following the use of a FUL instruction. The format of the ABS instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ABS	Blanks

The ABS operation is *effective only when the assembler is in the full mode* of operation, through the prior appearance of a FUL instruction. The effect of ABS is to cause any words remaining in the punch buffer to be written out and the next output to start on a new card, in the normal punching mode.

An ABS instruction will be ignored in a RELMOD assembly.

The TCD Pseudo-operation

The TCD (Transfer Card) pseudo-operation is used to delimit a core storage load when there is more than one load in an assembly. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	TCD	A non-complex expression

The TCD pseudo-operation performs the following function:

Text is produced that indicates to the Loader the end of a core storage load whose entry point is defined by the value of the expression in the variable field.

TCD is recommended for use only in a single ABSMOD assembly that does not call subroutines. The loading of subsequent parts of the deck must be accomplished by the programmer through the use of S.SLDR. The calling sequence to obtain subsequent core storage loads is:

```
TSX     S. SLDR, 4
MZE
```

The next instruction after the TCD should be a BCI identification word for the next memory load. This word will not be loaded and should be followed by an ORG pseudo-operation, as follows:

```
TCD     LCS
BCI     1, ID
ORG     SYMBOL
```

The ETC Pseudo-operation

The blank that separates the variable field of a symbolic card from the comments field is an end-of-card indicator rather than an end-of-variable-field indicator. The variable field of most instructions may be extended over more than one card by means of the ETC (Etcetera) pseudo-operation. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ETC	A series of subfields, separated by commas

The effect of this operation is to append its variable field as a continuation of the variable field of the previous card. As many as nine ETC cards may be used, provided no element of an expression is split between cards. For example, the instruction

```
TIX     NAME+1, 4, 1
```

could be written in the forms

```
TIX     NAME+1
ETC     ,4, 1
```

or

```
TIX     NAME+1, 4,
ETC     1
```

but could not be written as

```
TIX     NA
ETC     ME+1, 4, 1
```

Note that the following operations *may not* be followed by an ETC card:

ABS	EVEN	NOCRS	QUAL
BCI	FUL	NULL	REM
CONTRL	IRP	OPSYN	TITLE
DETAIL	LIST	ORGCRS	TTL
EJECT	LITORG	PCC	UNLIST
ENDM	LORG	PMC	USE
ENDQ			

If an ETC does follow one of these operations, it will be ignored.

The NULL Pseudo-operation

The NULL pseudo-operation defines a symbol. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	NULL	Any information

If there is a symbol in the name field, it is assigned the current value of the location counter; NULL has no other effect on the assembly.

The REM Pseudo-operation

The REM (Remarks) pseudo-operation is used to enter remarks into the assembly listing. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Any information, except that column 1 may not contain a dollar sign ($)	REM	Any information

In the listing, the contents of columns 8-10 (i.e., the operation field) will be replaced by blanks and the contents of the remainder of the card will be listed in full: The REM operation has no other effect on the assembly.

The REM card has been largely supplanted by remarks cards of the type with an asterisk in column 1. However, in a macro-definition, the variable field of an REM card will be scanned for substitution parameters, whereas the * remarks card will be completely ignored.

The END Pseudo-operation

The END pseudo-operation is used to signal the end of the symbolic deck. Thc format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	END	A non-complex expression or blanks

The effect of this operation is to terminate the assembly and to place the definition of the expression in the variable field in the Control Dictionary as the nominal starting point of the program segment. In a RELMOD assembly, the expression must be relocatable. If the variable field is blank, the load address is assumed in a RELMOD assembly; zero is assumed in an ABSMOD assembly.

The END instruction must be present and must be the last card of the symbolic deck.

Macro-operations

A *macro-operation* is a special type of pseudo-operation, the name and function of which are established by the programmer. The instruction that calls a macro-operation is a *macro-instruction.* The most significant property of a macro-instruction is that it generates n source card images, where n is usually greater than one. The programmer determines the card images to be generated by a given macro-instruction in a *macro-definition.*

The contents of the card images so generated are virtually unrestricted; a card within a macro-definition may contain any machine instruction, pseudo-instruction, or macro-instruction.

The sequence of instructions generated by a macro-instruction (i.e., the *macro-expansion*) is automatically inserted in the normal flow of the program and executed in-line, or serially, with the rest of the program each time the macro-instruction is used.

A macro-definition has three principal parts: a macro-definition heading card, prototype card images, and the terminating card.

The *macro-definition heading card* is the card containing the MACRO pseudo-instruction; the exact format of this instruction is given below. This card also contains the name of the macro-operation; this name becomes the operation code of the macro-instruction for that macro-operation. Finally, the macro-definition heading card contains a list of substitutable arguments; these are dummy parameters for those fields of a macro-operation that can be altered each time the macro-operation is used.

The *prototype card images* establish the actual operation, i.e., the instructions that are to be included and their sequence, and the position of those fields within the operation that are to be substitutable arguments and those that are to be text.

A prototype card image is a standard source card image having a name field, an operation field, and a variable field. It may also contain a comments field, but this field will not normally appear in the card image generated by the macro-instruction.

The fields (or subfields of the variable field) of a prototype card image may consist of the following:

1. *Text,* which is reproduced in the macro-expansion exactly as it appears in the prototype. A field or subfield is text if it does not appear in the substitutable argument list of the macro-definition heading card. Fields that are text are the fixed fields of the operation; they will remain the same no matter how the macro-operation is used and they must conform to the rules governing each of the instructions involved. For example, an operation field that is text must contain a valid operation code.
2. *Substitutable arguments,* which are also listed in the macro-definition heading card. Each substitutable argument consists of a string of from one to six characters.
3. *Punctuation (Special) Characters,* which delimit arguments and are reproduced in the macro-expansion as they appear in the prototype.

A macro-expansion consists of a reproduction of the prototype card images within a macro-definition, with text and punctuation characters exactly as they appear on the prototype card, but with the substitutable arguments replaced by parameters from the parameter list of the macro-instruction card that caused the expansion.

The *terminating card* of a macro-definition is the card containing the ENDM pseudo-instruction, discussed below.

Macro-definition Pseudo-operations

The MACRO Pseudo-operation

The MACRO pseudo-operation is used to define a macro-operation; the card bearing this instruction is the macro-definition heading card. The format of the MACRO instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
A name of up to six characters	MACRO	Up to 63 substitutable arguments (strings of not more than six characters), separated by special characters

The name in the name field is the operation code of the macro-instruction defined by the prototype that follows. It may be any valid symbol or may consist of all numeric characters. The name may be the same as any symbol appearing anywhere in the program. If it is the same as any other machine instruction code, pseudo-instruction code, or macro-instruction code, the new definition will replace the old definition.

The substitutable arguments in the variable field also appear in the prototype. Their appearance here

determines the order in which the parameters must appear in the macro-instruction. The substitutable arguments may be any valid symbols or may consist of all numeric characters. They may be separated on the heading card or delimited in the prototype by any of the following punctuation (special) characters:

= + − * / ( ) $, '

Hence, meaningful notation may be used in a macro-definition. For example, the substitutable argument list

```
QA1        MACRO    23, RATE, TIME, DIST,
                        QUSYM, SYM1, SYM2
```

could also be written

```
QA1        MACRO    23(RATE*TIME=DIST)
                        QUSYM$SYM1$SYM2
```

Consecutive punctuation characters are ignored and do not result in a substitutable argument. Parentheses *may not be used* as part of a substitutable argument.

Since these substitutable arguments are dummy names and have no effect on the execution of the program, they may be identical to strings of characters used elsewhere in the program, in location, operation, or variable fields, including the operation code for this or any other macro-instruction. Of course, they cannot be identical to strings within the prototype itself except those strings that are actually to be substituted. The programmer must exercise caution so that, in the prototype, fields intended as text are not confused with substitutable arguments, since every string of six or fewer characters, in any field, is compared with the substitutable argument list. Special care should be taken with the fields of BCI, VFD, DEC, or OCT pseudo-operations.

The effect of the MACRO pseudo-operation is to define the macro-operation by relating the name and the substitutable arguments to the prototype and entering the prototype into the *Macro Skeleton Table.*

The ENDM Pseudo-operation

The ENDM (End Macro) pseudo-operation terminates a macro-definition. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ENDM	A name of up to six characters or blanks

ENDM is used immediately following the last card of a prototype, and signals the end of the macro-definition. The name in the variable field is the name of the macro-instruction, as defined in the location field of the corresponding MACRO card. If the variable field is blank, all current macro-definitions are terminated (see "Nested Macro-Operations").

Prototype Card Images

The prototype of a macro-definition may include macro-instructions, pseudo-instructions, and machine instructions. Substitutable arguments may appear in any field or subfield of any prototype card. The substitutable arguments can be delimited by blanks or the following punctuation (special) characters:

= + − * / ( ) $, '

Except for the apostrophe ('), these characters are considered as text and will be reproduced in the expansion.

The prototype of a macro-definition may include macro-instructions that have not yet been defined; however, these lower-level macro-instructions must be defined before using a macro-instruction that generates the higher-level macro-operation. Circular definitions must be avoided; that is, a macro-operation A may not include a macro-instruction whose definition includes the macro-instruction for A. Also, a macro-operation may not include within its prototype its own macro-instruction.

Within a prototype, BCI, REM, and TTL cards are scanned in full for substitutable arguments. If the variable field of a BCI card begins with a character other than a comma in card column 12, a number, or a blank, the first subfield should be a substitutable argument for which a count will be substituted in the macro-instruction argument list.

If a blank is encountered before card column 72 in the variable field of a prototype card (except the cards BCI, REM, and TTL), it terminates the scan of the card, and any information to the right of the blank will not be included in the macro-definition.

Remarks cards of the type with an asterisk (*) in column 1 will appear in a macro-definition but not in the macro-expansion.

Defining a Macro-operation

Suppose a program contains the sequences of instructions in Figure 2.

```
        CLA     FEDTAX
        ADD     STATAX
        STO     TOTTAX
         .
         .
         .
SUBCOM  CLA     XSUB1
        ADD     YSUB1
        STO     ZSUB1
         .
         .
         .
        CLA     PART1
        ADD     PART2
        STO     TOTAL
```

Figure 2

The pattern of three instructions in Figure 2 might be identified by some name such as QSUM, and this name could then be defined as in Figure 3.

```
QSUM    MACRO   V1, V2, V3
        CLA     V1
        ADD     V2
        STO     V3
        ENDM    QSUM
```

Figure 3

The sequence in Figure 3 itself generates no words in the object program but constitutes the definition of the macro-operation QSUM, which is entered into the Macro Skeleton Table.

The first card is the macro-definition heading card, containing the name of the macro (QSUM) in the name field, the operation MACRO in the operation field, and the substitutable argument list in the variable field.

The next three cards are the prototype, in which V1, V2, and V3 are substitutable arguments. All of the other fields (CLA, ADD, STO) are text. The fifth card marks the end of the macro-definition.

The operation code ***, BSS, or ETC may not be used as the name of a macro. If BCI, ENDM, IRP, MACRO, REM, or TTL is used as the name of a macro, it may not be used within any macro-definition.

The Format of a Macro-instruction

Once a macro-operation has been defined, the macro-instruction that calls it is written as follows:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Symbol or blanks	Name of the macro-operation	The parameters, separated by commas or parentheses

If there is a symbol in the name field, it is assigned to a BSS 0 instruction that will be generated preceding the first card of the macro-expansion.

The argument list in the variable field contains the *parameters that are to replace the substitutable arguments* in the prototype. They must appear *in the same order* as the arguments they are to replace appeared in on the corresponding MACRO card. The parameter list can be extended using ETC cards (see "The ETC Pseudo-Operation").

It is not necessary to restrict the length of a parameter to be substituted to six characters. An entire instruction may be inserted into any field; no blank will be inserted following a name field longer than six characters, and the operation field, if any, will follow immediately. The only restriction on the length of a single parameter is that it be no more than 61 characters and that it appear entirely on a single card.

The parameters of a macro-instruction parameter list are separated either by commas or by being enclosed in a pair of parentheses. Between two parameters, a single comma following a right parenthesis, or a single comma preceding a left parenthesis, is redundant and may be omitted; it *does not result* in a null parameter. In this case, a null parameter is indicated by two consecutive commas; if a zero parameter is desired, an explicit zero must appear in the parameter list. For example, the following parameter lists are equivalent and will generate the same expansion:

```
ALPHA,BETA,GAMMA
ALPHA,(BETA),GAMMA
ALPHA(BETA)GAMMA
```

If, however, a null parameter is desired as the first or last parameter of the list, this is indicated by a single comma preceding a left parenthesis or following a right parenthesis, respectively, as shown below:

```
,(BETA)GAMMA
ALPHA(BETA),
```

A pair of parentheses surrounding a string in a macro-instruction parameter list signifies that *everything within the parentheses,* including blanks and special characters, is to be substituted for the corresponding argument in the macro-definition prototype. The parentheses will be removed during the expansion.

If, in a macro-instruction parameter list, parentheses are to be included as part of a parameter, they must be enclosed within another pair of parentheses, which will be removed in the expansion of the macro-operation. In any case, pairs of parentheses must be balanced.

A macro-operation must be defined in a program *prior to the use* of the macro-instruction. Once the operation QSUM has been defined as in Figure 3, each of the sequences in Figure 2 could be replaced by macro-instructions as in Figure 4.

```
         QSUM     FEDTAX, STATAX, TOTTAX
          .
          .
          .
SUBCOM   QSUM     XSUB1, YSUB1, ZSUB1
          .
          .
          .
         QSUM     PART1, PART2, TOTAL
```

Figure 4

Note that SUBCOM will be assigned as the location symbol of a BSS 0 instruction preceding the first instruction of that macro-expansion, i.e., the CLA XSUB1 instruction in Figure 2.

If the order of the substitutable arguments in the macro-definition heading card of Figure 3 were changed to, for example, V3, V1, V2, then the macro-instructions in Figure 5 would produce the same macro-expansions.

```
         QSUM     TOTTAX, FEDTAX, STATAX
          .
          .
          .
SUBCOM   QSUM     ZSUB1, XSUB1, YSUB1
          .
          .
          .
         QSUM     TOTAL, PART1, PART2
```

Figure 5

In the example given in Figure 3, the substitutable arguments all appeared in the address fields of the prototype and were replaced by symbols in the macro-expansion. However, substitutable arguments may appear in the name field, in the operation field, or in any of the subfields of the variable field. Moreover, the arguments may be replaced by any appropriate character strings.

For example, a macro-definition can be written as in Figure 6.

```
QPOLY    MACRO    COEFF, LOOP, DEG, T, OP
         AXT      DEG, T
         LDQ      COEFF
LOOP     FMP      GAMMA
         OP       COEFF+DEG+1, T
         STO      TEMP
         LDQ      TEMP
         TIX      LOOP, T, 1
         ENDM     QPOLY
```

Figure 6

In the example in Figure 6, mnemonic character strings have been used to represent the substitutable arguments. Notice that LOOP appears in a name field, OP in an operation field, and that COEFF and DEG appear as symbols and as elements within expressions in address subfields. Notice also that GAMMA and TEMP are text, i.e., symbols and not substitutable arguments, and presumably are defined elsewhere in the program.

Any use of QPOLY should be accompanied by a parameter list of appropriate substitutions for the substitutable arguments. For example, LOOP should be replaced by a symbol, and OP by a valid operation code. A QPOLY macro-instruction might be written:

```
X015     QPOLY    C1-4, FIRST, 5, 4, FAD
```

This macro-instruction would cause the eight card images to be generated as in Figure 7:

```
X015     BSS      0
         AXT      5, 4
         LDQ      C1-4
FIRST    FMP      GAMMA
         FAD      C1-4+5+1, 4
         STO      TEMP
         LDQ      TEMP
         TIX      FIRST, 4, 1
```

Figure 7

In the macro-expansion in Figure 7, the symbol X015 is assigned to a BSS 0 instruction preceding the first instruction, and each of the substitutable arguments is replaced by the corresponding parameter in the macro-instruction parameter list. The expression arising from the prototype address COEFF+DEG+1 reduces to C1+2.

The following example illustrates the insertion of an entire instruction in a single field of a prototype. Given the macro-definition

```
XYZ      MACRO    A, B, C
         CLA      A
         B
         STO      C
         ENDM     XYZ
```

then the macro-instruction (where b represents a blank)

```
SUM      XYZ      ALPHA(ADDbbbbbBETA)
                  GAMMA
```

would result in the expansion

```
SUM      CLA      ALPHA
         ADD      BETA
         STO      GAMMA
```

Linking Partial Subfields

The special character ' (apostrophe) is used to concatenate (link) partial subfields. It is possible to create a single subfield combining arguments and text, since the apostrophe delimits an argument in the macro-definition prototype but is not itself included in the macro-expansion. If apostrophes are used in the vari-

able fields of BCI, REM, or TTL prototype cards to concatenate fields, they must be used in pairs and must surround the substitutable argument. On other types of prototype cards, only one apostrophe is necessary to indicate concatenation.

The location field of a prototype card may not contain more than six characters, including text, substitutable arguments, and apostrophe(s). The operation field may not contain more than seven characters.

For example, given the macro-definition

```
MAC1        MACRO   X,Y,Z,...
            SX'X    ADDR'Y,Z
              .
              .
              .
            ENDM    MAC1
```

then the first instruction generated by the macro-instruction

```
            MAC1    A,1,2,...
```

would be

```
            SXA     ADDR1,2
```

The first instruction generated by the macro-instruction

```
            MAC1    D,2,2,...
```

would be

```
            SXD     ADDR2,2
```

Given the macro-definition

```
ALPHA       MACRO   A,B,C,
            BCI     A,bb'B'bERROR.bCONDI-
                      TION'C'bIGNORED.
            ENDM    ALPHA
```

where b represents a blank, then the macro-instruction

```
            ALPHA   6,(FIELD),
```

would cause the following card to be generated:

```
            BCI     6, bbFIELDbERROR. bCON-
                      DITIONbIGNORED.
```

Qualification Within Macro-operations

The character $ delimits an argument in the prototype and is itself included as text within the macro-expansion. This character may be used to indicate qualification either in the prototype or in the macro-instruction parameter list.

Any qualification in effect at the time a macro-instruction is encountered will be used for symbols defined in that macro-expansion. If a qualification symbol is required within a macro-operation prototype, it may be a substitutable argument, as may the symbols it qualifies.

If a macro-operation containing a qualification section falls within the range of a qualification section in the program, the rules for nested qualification apply when referring to symbols defined within the macro-expansion.

Nested Macro-operations

Macro-operation definitions may be nested; i.e., a macro-operation definition may be entirely included within the range of another (higher-level) macro-operation definition.

Lower-level macro-operations are not defined until all higher-level macro-operations within which they are nested have been expanded. Therefore, the macro-instruction of a lower-level macro-operation cannot be used until the macro-instructions of all higher-level macro-operations have been expanded.

When the assembler encounters an outer MACRO card, it defines the higher-level macro-operation and enters it into the Macro Skeleton Table. Then, when the higher-level macro-instruction is used, the expansion of that macro-operation is inserted in-line in the program, with its substitute arguments replaced by parameters from the macro-instruction parameter list. At this point, the assembler encounters the MACRO card for the next lower-level macro-operation, defines that macro-operation, and enters it into the Macro Skeleton Table.

Each time a higher-level macro-instruction is used, the next lower-level macro-operation will be defined and an entry will be made in the Macro Skeleton Table. Thus, if the name of the lower-level macro-operation is a substitutable argument in the higher-level macro-definition, each use of the higher-level macro-instruction causes additional macro-operations to be defined; if the name of the lower-level macro-operation is text in the higher-level macro-definition, each use of the higher-level macro-instruction causes the lower-level macro-operation to be redefined. Defining new macros or redefining existing ones with a single instruction is the most common use of the nested macro-operation facility.

For example, given the sequence

```
MAC1        MACRO   MAC2, ALPHA, BETA,
                      GAMMA, DELTA
MAC2        MACRO   ALPHA
            BETA    A
            GAMMA   B
            DELTA   C
            ENDM    MAC2
            ENDM    MAC1
```

then the instruction

```
            MAC1    ABC, (A, B, C), CLA,
                      ADD, STO
```

would generate

```
ABC         MACRO   A, B, C
            CLA     A
            ADD     B
            STO     C
            ENDM    ABC
```

and ABC is defined as a macro-operation with A, B, and C as substitutable arguments, and CLA, ADD, and STO as text.

The instruction

```
        MAC1    XYZ, (OP1, OP2), OP1,
                  OP2, STO
```

would generate

```
XYZ     MACRO   OP1, OP2
        OP1     A
        OP2     B
        STO     C
        ENDM    XYZ
```

and XYZ is defined as a macro-operation with OP1 and OP2 as substitutable arguments, and A, B, C, and STO as text.

There is no significant limit to the depth of the nesting allowed.

DUP With Macro-operations

A macro-operation may generate a DUP. Subsequent card images generated by the same macro are counted to determine the range of the DUP. If another macro-operation is encountered during this generation, only the macro-operation card image is counted (i.e., none of the images generated at lower levels by the macro are counted). If the end of the range has not been reached when the macro-generation stops, counting continues at the source level (or a higher macro level if the DUP was generated by an inner level macro). Once a higher level has been encountered, counting is not resumed at any subsequent lower levels.

1. Example of a macro falling within the range of a DUP statement:

Given the macro-definition

```
MACA    MACRO   . . . . .
        InstA   . . . . .
        InstB   . . . . .
        ENDM    MACA
```

then the instructions

```
        DUP     2, 2
        InstC   . . . . .
        MACA    . . . . .
```

would generate

		count	*iteration*
InstC	. . . . .	1	1
InstA	. . . . .	2	
InstB	. . . . .		
InstC	. . . . .	1	2
InstA	. . . . .	2	
InstB	. . . . .		

2. Example of a DUP statement within a macro-definition:

Given the macro-definition

```
MACB    MACRO   . . . . .
        InstB   . . . . .
        DUP     1, 3
        InstC   . . . . .
        InstD   . . . . .
        ENDM    MACB
```

then the instructions

```
        InstA   . . . . .
        MACB    . . . . .
        InstE   . . . . .
```

would generate

		count	*iteration*
InstA	. . . . .	—	—
InstB	. . . . .	—	—
InstC	. . . . .	1	1
InstC	. . . . .	1	2
InstC	. . . . .	1	3
InstD	. . . . .	—	—
InstE	. . . . .	—	—

3. Example of a DUP statement within a macro-definition that contains a call to a lower level:

Given the macro-definitions

```
MACA    MACRO   . . . . .
        InstA   . . . . .
        InstB   . . . . .
        ENDM    MACA
MACB    MACRO   . . . . .
        InstC   . . . . .
        DUP     2, 2
        MACA    . . . . .
        InstD   . . . . .
        InstE   . . . . .
        ENDM    MACB
```

then the instruction

```
        MACB    . . . . .
```

would generate

		count	*iteration*
InstC	. . . . .	—	—
InstA	. . . . .	1	1
InstB	. . . . .		
InstD	. . . . .	2	
InstA	. . . . .	1	2
InstB	. . . . .		
InstD	. . . . .	2	
InstE	. . . . .	—	—

4. Example of a DUP statement within a macro-definition; the range of the DUP being longer than the range of the macro:

Given the macro-definition

```
MACA    MACRO   . . . . .
        InstA   . . . . .
        DUP     2, 2
        InstB   . . . . .
        ENDM    MACA
```

then the instructions

```
        MACA    . . . . .
        InstC   . . . . .
```

would generate

		count	*iteration*
InstA	. . . . .	—	—
InstB	. . . . .	1	1
InstC	. . . . .	2	
InstB	. . . . .	1	2
InstC	. . . . .	2	

Macro-related Pseudo-operations

The macro-related pseudo-operations (IRP, NOCRS, ORGCRS) are provided to extend the facilities of macro-operations. They have no significance except in relation to macro-operations.

The IRP Pseudo-operation

The IRP (Indefinite Repeat) pseudo-operation is used to repeat a sequence of card images within a macro-operation, varying one argument in the sequence each time the sequence is repeated. IRP can only be used within a macro-operation definition; it is undefined outside of macro-operations. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	IRP	Depends on whether the IRP is to initiate or terminate the sequence to be repeated, as follows: 1. To initiate a sequence, a single substitutable argument 2. To terminate a sequence, blanks

For example, the instruction

```
        IRP     ARG
```

defines the beginning of such a sequence

```
        IRP
```

defines the end of the sequence, and the variable field ARG governs the iterations of the card images to be generated within the sequence.

The macro-instruction that causes the macro-operation to be expanded should contain in its parameter list, for the position corresponding to the substitutable argument ARG, a series of subparameters, separated by commas and enclosed within parentheses. All of the subparameters must appear on a single card (either the macro-instruction card itself or a subsequent ETC card). Each subparameter will cause the sequence to be repeated once, with ARG replaced by the subparameter. If only one subparameter is given, the sequence will appear once; if the subparameter list is null, the whole sequence will be skipped. If there is a blank in this subfield, succeeding subparameters will be ignored.

For example, given the macro-definition

```
XYZ     MACRO   ARG,B
        IRP     ARG
        CLA     ARG
        ADD     B
        STO     ARG
        IRP
        ENDM    XYZ
```

then the instruction

```
        XYZ     (J,K,L),CONST
```

would generate the following instructions:

```
CLA     J       First sequence,
ADD     CONST   with subparam-
STO     J       eter J
CLA     K       Second sequence,
ADD     CONST   with subparam-
STO     K       eter K
CLA     L       Third sequence,
ADD     CONST   with subparam-
STO     L       eter L
```

The variable field of an initial IRP must be a single substitutable argument; if it is not, the IRP will be ignored and the sequence of instructions enclosed within the IRP statements will be generated once.

If the substitutable argument does not appear in the range of the IRP statements, the iterations will be identical, their number depending on the number of subparamcters given.

An IRP cannot occur explicitly within the range of another IRP; such a nested pair causes the termination of the first range and the opening of another range. However, a macro-instruction within the range of an IRP may itself contain pairs of IRP operations, and nested IRP ranges may be created in this way.

Created Symbols

If parameters are omitted from the end of the parameter list of a macro-instruction, the assembler automatically supplies symbols to fill out the parameter list. These symbols, called *created symbols*, are of the form

```
. . 0001
. . 0002
. . 0003
  .
  .
  .
. . nnnn
```

A parameter indicated by a comma is treated as null; created symbols are supplied only at the end of the parameter list as shown in the following example:

Given the macro-definition heading card

```
ALPHA   MACRO   A,B,C
```

then the instruction

```
        ALPHA   X,
```

will cause: each appearance of the substitutable argument A to be replaced by X; each appearance of the substitutable argument B to be omitted, since the parameter is explicitly null; and each appearance of the substitutable argument C to be replaced by a symbol of the form . . nnnn, created to replace the omitted parameter at the end of the macro-instruction parameter list.

The pseudo-operations NOCRS and ORGCRS are used to control the creation of symbols in macro-instructions following the NOCRS or ORGCRS.

The NOCRS Pseudo-operation

The NOCRS (No Created Symbols) pseudo-operation is used to suppress the creation of symbols for macro-instruction parameter lists. The format of this instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	NOCRS	Blanks

Note that creation of symbols is the normal mode of the assembler, and if it is desired to suppress their creation, the NOCRS pseudo-operation must be used.

The ORGCRS Pseudo-operation

The ORGCRS (Origin Created Symbols) pseudo-operation is used to resume the creation of symbols if they have been suppressed by a NOCRS instruction. The format of the ORGCRS instruction is:

NAME FIELD	OPERATION FIELD	VARIABLE FIELD
Blanks	ORGCRS	Blanks

Creation of symbols will be resumed starting with the next symbol to be created when the NOCRS instruction was encountered.

Appendix A. 7040/7044 Machine Operations, Special Operations, Prefix Codes, and IOCS Operations

This appendix contains complete lists of all 7040/7044 machine operations, special operations, prefix codes, and IOCS operations.

INSTRUCTION ASSEMBLY

The assembly of machine operations, special operations, prefix codes, and IOCS operations involves the following functions:

1. If there is a symbol in the name field, this symbol is given the value of the next location to be assigned by the assembler when the instruction is encountered.
2. The operation code is translated into a 36-bit binary instruction word. Note that the bits that determine the operation may occupy positions in the prefix, decrement, and address portions of the internal binary word.
3. If indirect addressing has been specified, the appropriate flag bits are inserted.
4. If an address subfield is present, the expression in this subfield is evaluated.* The 15-bit result is combined by a logical OR with the rightmost 15 bits of the instruction word (i.e., the address portion).
5. If a tag subfield is present, the expression in this subfield is evaluated and the rightmost three bits of the result are combined by a logical OR with the tag portion of the instruction word. If there is an expression in this subfield, it must be absolute.
6. If a decrement subfield is present, the expression in this subfield is evaluated* and the n low-order bits of the result (where n is the number of decrement bits as given in the table below) are combined by a logical OR with the decrement portion of the instruction word.
7. The 36-bit instruction that results is assigned to the next location to be assigned by the assembler.

COLUMN HEADINGS

The column headings used in this appendix are as follows:

Mnemonic: This column gives the BCD operation code of the instruction.

Ind: An I in this column indicates that indirect addressing is permissible.

Subfields: The column headings ADDR, TAG, and DECR stand for the address, tag, and decrement subfields of the variable field. The coding in these columns is:

Y	Address or number
T	Tag
D	Decrement
B	Blank
P	Character position
C	Count
I	Interface

A slash is used to indicate alternatives (e.g., T/B indicates that a tag field is permissible but not required and may be blank). The number in parentheses following the decrement indicates the number of bits in the decrement; in some cases, this number is given in a separate column headed DECR BITS.

Min: This column gives the minimum number of subfields required for the instruction. For example, if the minimum is given as 2, the assembler will expect two subfields, and an error will be indicated if there is less than two. (Note that these subfields may be null; see "Symbolic Instructions.")

Max: This column gives the maximum number of subfields allowed for the instruction.

Octal Prefix: For prefix codes, this column gives the octal configuration of the bits that will be assembled in the prefix portion of the word.

Assembles As: For special operations, this column indicates how the operation code will be assembled.

Opt: This column indicates the option set that the instruction belongs to. This column is coded as follows:

CODE	OPERATION SET
0	Channel A and Basic Operations
1	Extended Machine Operations
2	Single-Precision Floating-Point Operations
3	Double-Precision Floating-Point Operations
4	Storage Protection Operations
B	Channel B Operations
C	Channel C Operations
D	Channel D Operations
E	Channel E Operations

Machine Operations

MNE-		SUBFIELDS					
MONIC	IND	ADDR	TAG	DECR	MIN	MAX	OPT
ACL	I	Y	T/B	B(4)	1	2	0
ADD	I	Y	T/B	B(4)	1	2	0
ALS		Y	T/B	B(6)	1	2	0
ANA	I	Y	T/B	B(4)	1	2	0
ARS		Y	T/B	B(6)	1	2	0
AXT		Y	T	B(6)	2	2	1
BSR		Y	T/B	I/B(4)	1	3	0
CAL	I	Y	T/B	B(4)	1	2	0
CAS	I	Y	T/B	B(4)	1	2	0

*Note that certain types of expressions are evaluated by the Loader at Load time; however, the effect is the same as the description above.

MNE-	SUBFIELDS							MNE-	SUBFIELDS						
MONIC	IND	ADDR	TAG	DECR	MIN	MAX	OPT	MONIC	IND	ADDR	TAG	DECR	MIN	MAX	OPT
CCS	I	Y	T/B	P(4)	3	3	1	RDCE		B	T/B	B(6)	0	2	E
CHS		B	T/B	B(6)	0	2	0	RDS		Y	T/B	I/B(3)	1	3	0
CLA	I	Y	T/B	B(4)	1	2	0	REW		Y	T/B	I/B(6)	1	3	0
CLS	I	Y	T/B	B(4)	1	2	0	RPM		B	B	B(6)	0	2	4
COM		B	T/B	B(6)	0	2	0	RQL		Y	T/B	B(6)	1	2	0
CTR		Y	T/B	I(3)	1	3	0	RUN		Y	T/B	I/B(6)	1	3	0
DCT		B	T/B	B(6)	0	2	0	SAC	I	Y	T/B	P(4)	3	3	1
DFAD	I	Y	T/B	B(4)	1	2	3	SCHA	I	Y	T/B	B(0)	1	2	0
DFDP	I	Y	T/B	B(4)	1	2	3	SCHB	I	Y	T/B	B(0)	1	2	B
DFMP	I	Y	T/B	B(4)	1	2	3	SCHC	I	Y	T/B	B(0)	1	2	C
DFSB	I	Y	T/B	B(4)	1	2	3	SCHD	I	Y	T/B	B(0)	1	2	D
DVP	I	Y	T/B	B(4)	1	2	0	SCHE	I	Y	T/B	B(0)	1	2	E
ENB	I	Y	T/B	B(4)	1	2	0	SEN		Y	T/B	I(6)	1	3	0
ENK		B	T/B	B(6)	0	2	0	SLFA		B	T/B	B(6)	0	2	0
ETTA		B	T/B	B(6)	0	2	0	SLNA		B	T/B	B(6)	0	2	0
ETTB		B	T/B	B(6)	0	2	B	SLW	I	Y	T/B	B(4)	1	2	0
ETTC		B	T/B	B(6)	0	2	C	SPM	I	Y	T/B	C(4)	1	2	4
ETTD		B	T/B	B(6)	0	2	D	SSLB	I	Y	T/B	B(4)	1	2	B
ETTE		B	T/B	B(6)	0	2	E	SSLC	I	Y	T/B	B(4)	1	2	C
FAD	I	Y	T/B	B(4)	1	2	2	SSLD	I	Y	T/B	B(4)	1	2	D
FDP	I	Y	T/B	B(4)	1	2	2	SSLE	I	Y	T/B	B(4)	1	2	E
FMP	I	Y	T/B	B(4)	1	2	2	SSP		B	T/B	B(6)	0	2	0
FSB	I	Y	T/B	B(4)	1	2	2	STA	I	Y	T/B	B(4)	1	2	0
HPR		Y/B	B	B(6)	0	2	0	STD	I	Y	T/B	B(4)	1	2	0
ICT		B	T/B	B(6)	0	2	0	STL	I	Y	T/B	B(4)	1	2	0
IORD		Y	B	C(15)	3	3	0	STO	I	Y	T/B	B(4)	1	2	0
IOT		B	T/B	B(6)	0	2	0	STQ	I	Y	T/B	B(4)	1	2	0
LAC		Y	T	B(6)	2	2	1	STR		B	B	B(6)	0	2	0
LAS	I	Y	T/B	B(4)	1	2	0	STZ	I	Y	T/B	B(4)	1	2	0
LBT		B	T/B	B(6)	0	2	0	SUB	I	Y	T/B	B(4)	1	2	0
LDC		Y	T	B(6)	2	2	1	SWT		Y	T/B	B(6)	1	2	0
LDQ	I	Y	T/B	B(4)	1	2	0	SXA		Y	T	B(6)	2	2	1
LGL		Y	T/B	B(6)	1	2	0	SXD		Y	T	B(6)	2	2	1
LGR		Y	T/B	B(6)	1	2	0	TCOA	I	Y	T/B	B(4)	1	2	0
LLS		Y	T/B	B(6)	1	2	0	TCOB	I	Y	T/B	B(4)	1	2	B
LRS		Y	T/B	B(6)	1	2	0	TCOC	I	Y	T/B	B(4)	1	2	C
LXA		Y	T	B(6)	2	2	1	TCOD	I	Y	T/B	B(4)	1	2	D
LXD		Y	T	B(6)	2	2	1	TCOE	I	Y	T/B	B(4)	1	2	E
MIT	I	Y	T/B	B(0)	1	2	1	TDOA	I	Y	T/B	I(4)	3	3	0
MPY	I	Y	T/B	B(4)	1	2	0	TEFA	I	Y	T/B	B(4)	1	2	0
MSM	I	Y	T/B	B(0)	1	2	1	TEFB	I	Y	T/B	B(4)	1	2	B
MSP	I	Y	T/B	B(0)	1	2	1	TEFC	I	Y	T/B	B(4)	1	2	C
ORA	I	Y	T/B	B(4)	1	2	0	TEFD	I	Y	T/B	B(4)	1	2	D
PAC		B	T	B(6)	2	2	1	TEFE	I	Y	T/B	B(4)	1	2	E
PAX		B	T	B(6)	2	2	1	TIX		Y	T	D(15)	3	3	1
PBT		B	T/B	B(6)	0	2	0	TMI	I	Y	T/B	B(4)	1	2	0
PCS	I	Y	T/B	P(4)	3	3	1	TMT		Y	T	B(6)	1	2	1
PDC		B	T	B(6)	2	2	1	TNX		Y	T	D(15)	3	3	1
PDX		B	T	B(6)	2	2	1	TNZ	I	Y	T/B	B(4)	1	2	0
PLT	I	Y	T/B	B(0)	1	2	1	TOV	I	Y	T/B	B(4)	1	2	0
PRD		Y	T/B	I/B(3)	1	3	0	TPL	I	Y	T/B	B(4)	1	2	0
PSLB	I	Y	T/B	B(4)	1	2	B	TRA	I	Y	T/B	B(4)	1	2	0
PSLC	I	Y	T/B	B(4)	1	2	C	TRCA	I	Y	T/B	B(4)	1	2	0
PSLD	I	Y	T/B	B(4)	1	2	D	TRCB	I	Y	T/B	B(4)	1	2	B
PSLE	I	Y	T/B	B(4)	1	2	E	TRCC	I	Y	T/B	B(4)	1	2	C
PWR		Y	T/B	I/B(6)	1	3	0	TRCD	I	Y	T/B	B(4)	1	2	D
PXA		B	T	B(6)	2	2	1	TRCE	I	Y	T/B	B(4)	1	2	E
PXD		B	T	B(6)	2	2	1	TRP	I	Y	T/B	B(4)	1	2	0
RCHA	I	Y	T/B	B(0)	1	2	0	TRT	I	Y	T/B	B(4)	1	2	0
RCHB	I	Y	T/B	B(0)	1	2	B	TSL	I	Y	T/B	B(4)	1	2	0
RCHC	I	Y	T/B	B(0)	1	2	C	TSX		Y	T	B(4)	2	2	1
RCHD	I	Y	T/B	B(0)	1	2	D	TXH		Y	T	D(15)	3	3	1
RCHE	I	Y	T/B	B(0)	1	2	E	TXI		Y	T	D(15)	3	3	1
RCT		B	T/B	B(6)	0	2	0	TXL		Y	T	D(15)	3	3	1
RDCA		B	T/B	B(6)	0	2	0	TZE	I	Y	T/B	B(4)	1	2	0
RDCB		B	T/B	B(6)	0	2	B	UFA	I	Y	T/B	B(4)	1	2	2
RDCC		B	T/B	B(6)	0	2	C	UFM	I	Y	T/B	B(4)	1	2	2
RDCD		B	T/B	B(6)	0	2	D	UFS	I	Y	T/B	B(4)	1	2	2
								VDP	I	Y	T/B	C(6)	3	3	0

MNE-		SUBFIELDS					
MONIC	IND	ADDR	TAG	DECR	MIN	MAX	OPT
VLM	I	Y	T/B	C(6)	3	3	0
VMA	I	Y	T/B	C(6)	3	3	0
WBT		Y	T/B	I/B(3)	1	3	0
WEF		Y	T/B	I/B(6)	1	3	0
WRS		Y	T/B	I/B(6)	1	3	0
XEC	I	Y	T/B	B(0)	1	2	0

Prefix Codes

In writing subroutine calling sequences, it is often necessary to specify parameters in each of the four sections of a binary word — prefix, decrement, tag, and address. The decrement, tag, and address may be specified in the variable field. (Of course, they must be given in the order: address, tag, decrement.) To enable the programmer to specify the value of the prefix bits, the following codes have been provided:

MNEMONIC	MEANING	OCTAL PREFIX	IND	DECR BITS	MIN	MAX
Blank	Zero	0		15	0	3
PZE or Zero	Plus Zero	0	I	15	0	3
PON or ONE	Plus One	1	I	15	0	3
PTW or TWO	Plus Two	2	I	15	0	3
PTH or THREE	Plus Three	3	I	15	0	3
MZE	Minus Zero	4	I	15	0	3
FOR or FOUR	Four	4	I	15	0	3
MON	Minus One	5	I	15	0	3
FVE or FIVE	Five	5	I	15	0	3
MTW	Minus Two	6	I	15	0	3
SIX	Six	6	I	15	0	3
MTH	Minus Three	7	I	15	0	3
SVN or SEVEN	Seven	7	I	15	0	3

Note that certain operations may generate instruction words identical to those generated by the prefix codes above (e.g., PTH, THREE, and TXH all generate the same instruction word).

Special Operations

The following special operations are supplied to provide operation codes for frequently-occurring conditions:

MNE-		ASSEM-		DECR		
MONIC	MEANING	BLES AS	IND	BITS	MIN	MAX
. . .	Operation code to be inserted during execution	PZE	I	15	0	3
***	Operation code to be inserted during execution	PZE		15	0	3
	Blank operation code	PZE		15	0	3
BRA	Branch	TXL		15	1	3
BRN	No operation setting of a branch/no operation switch	TXH		15	1	3
MSE	Minus sense	$(-0760)_8$		6	1	2
NOP	No operation	AXT		6	0	1
PSE	Plus sense	$(+0760)_8$		6	1	2
ZAC	Zero accumulator	PXD		6	0	1
ZSA	Zero storage address	SXA		6	1	1
ZSD	Zero storage decrement	SXD		6	1	1

IOCS Operations

7040/7044 MAP recognizes the 7040/7044 IOCS operations given below, and will accept them in source language input statements. They are described in detail in the IBM publication, *IBM 7040/7044 Operating System (16/32K): Input/Output Control System,* Form C28-6309, and, therefore, are only listed here.

MNEMONIC	ASSEMBLES AS	DECR BITS	MIN	MAX
ILCLS	MZE	15	3	3
ILFER	PTW	15	3	3
ILOPN	PON	0	3	3
ILRSW	PTH	15	3	3
IODER	0003	0	1	2
IODLY	0004	0	0	0
IORBI	−0020	0	1	2
IORBP	−0220	0	1	2
IORBR	−0320	0	1	2
IORBS	−0120	0	1	2
IORDI	−0000	0	1	2
IORDP	−0200	0	1	2
IORDR	−0300	0	1	2
IORDS	−0100	0	1	2
IOSKP	0001	0	1	2
IOSNS	0105	0	1	2
IOTPI	−0001	0	0	2
IOWBI	0020	0	1	2
IOWBP	0220	0	1	2
IOWBR	0320	0	1	2
IOWBS	0120	0	1	2
IOWDI	0000	0	1	2
IOWDP	0200	0	1	2
IOWDR	0300	0	1	2
IOWDS	0100	0	1	2
IOWEF	0002	0	0	0

Indirect addressing is not indicated in an IOCS operation by placing an asterisk after the operation code.

System Macro

Checkpoint: The CHKPNT macro-instruction generates coding for checkpoint initialization. This is described in detail in the IBM publication, *IBM 7040/7044 Operating System (16/32K): Input/Output Control System,* Form C28-6309.

IBM 1301 Disk and 7320 Drum Orders

The 7040/7044 Macro Assembly Program (IBMAP) recognizes the following Disk and Drum Orders, described in the publications *IBM 1301 Models 1 and 2 Disk Storage, and IBM 1302 Models 1 and 2 Disk Storage with IBM 7040/7044 Data Processing System,* Form A22-6768, and *IBM 7320 Drum Storage with 7040/7044 System,* Form A22-6793:

MNEMONIC	MEANING	ASSEMBLES AS	REQUIRED SUBFIELDS
DNOP	No Operation	0000	None
DREL	Release	0004	None
DEBM	Eight-Bit Mode	0010	None
DSBM	Six-Bit Mode	0011	None
DSEK	Seek	1000	a, t
DVSR	Prepare to Verify (Single Record)	1002	a, t, r
DWRF	Prepare to Write Format	1003	a, t

MNEMONIC	MEANING	ASSEMBLES AS	REQUIRED SUBFIELDS
DVTN	Prepare to Verify (Track Without Address)	1004	a, t
DVCY	Prepare to Verify (Cylinder Mode)	1005	a, t
DWRC	Prepare to Write Check	1006	a, t, r
DSAI	Set Access Imperative	1007	a
DVTA	Prepare to Verify (Track with Address)	1010	a, t
DVHA	Prepare to Verify (Home Address)	1011	a, t

The symbolic representation of a Disk or Drum Order should be of the form:

name dord a, t, r

1. name — any symbol or blanks
2. dord — the Disk or Drum Order operation code
3. a — any symbolic expression that represents the access mechanism and module. It is evaluated and converted to BCD, the low-order two characters being inserted into the instruction.
4. t — any symbolic expression that represents the track. It is evaluated and converted to BCD, the low-order four characters being inserted into the instruction.
5. r — two alphameric characters that represent the record to be selected. If this field is used, two and only two characters must be specified.

The a, t, and r subfields may be present even if not required. The operation code will assemble as two BCD characters; hence, the order will consist of from two to ten BCD characters. These BCD characters are left-justified in two consecutive words. Any unused portions of these words contain zeros.

Examples: In these examples assume that the following relationships hold:

```
ALPHA   EQU   3
BETA    EQU   8
GAMMA   EQU   3329
DELTA   EQU   35
```

a. DVTN ALPHA*BETA, GAMMA
Assembles as:
1004 0204 0303
0211 0000 0000

b. DWRC ALPHA+BETA, GAMMA, BA
Assembles as:
1006 0101 0303
0211 2221 0000
Notice that the selected record, BA, is converted directly to BCD as 2221.

c. DWRC ALPHA*DELTA, GAMMA*BETA, AA
Assembles as:
1006 0005 0606
0302 2121 0000
Notice that ALPHA*DELTA is 105, but only 05 (the low-order two characters) is assembled. Similarly, GAMMA*BETA is 26632, but only 6632 (the low-order four characters) is assembled.

d. DWRC 10,1381,BB
Assembles as:
1006 0100 0103
1001 2222 0000

To include the Disk and Drum Orders in the MAP assembler, the user must reassemble MAP. The publication *IBM 7040/7044 Operating System (16/32K): Systems Programmers Guide*, Form C28-6339, explains this procedure in the section called "Macro Assembly Program."

Appendix B. The MAP Internal 7040 Character Code (9-Code)

The following table lists the MAP graphics with the corresponding 7040 internal character code (9 code) and the standard IBM card code (H code).

GRAPHIC	9 CODE (OCTAL)	H CODE (CARD CODE)
blank	60	blank (no punch)
0	00	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
A	21	12-1
B	22	12-2
C	23	12-3
D	24	12-4
E	25	12-5
F	26	12-6
G	27	12-7
H	30	12-8
I	31	12-9
J	41	11-1

GRAPHIC	9 CODE (OCTAL)	H CODE (CARD CODE)
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
S	62	0-2
T	63	0-3
U	64	0-4
V	65	0-5
W	66	0-6
X	67	0-7
Y	70	0-8
Z	71	0-9
+ (plus)	20	12
− (minus)	40	11
/ (slash)	61	0-1
= (equals)	13	8-3
' (apostrophe)	14	8-4
. (period)	33	12-8-3
) (right parenthesis)	34	12-8-4
$ (dollar sign)	53	11-8-3
* (asterisk)	54	11-8-4
, (comma)	73	0-8-3
((left parenthesis)	74	0-8-4

Appendix C. Example of Assembly Output

The output listing of an IBMAP assembly has four principal parts: the Control Dictionary, the assembled text, the cross-reference dictionary, and the error messages.

Control Dictionary Listing

Figure 8 illustrates a typical Control Dictionary listing. The first column on the left contains the number of the Control Dictionary entry. The second column contains the octal representation of the binary entry. The remaining three columns are generated by the assembler for documentation purposes: the third column is the external identification for the entry, the fourth column indicates the type of entry, and the last column gives the deck information.

Assembled Text Listing

Figure 9 illustrates a segment of a listing of assembled text. There are two types of text output: file text and instruction text.

The file text, if any, precedes the instruction text and is used by the Loader to generate file control blocks. The first column is the octal representation of the binary input to the Loader. The next column contains the binary relocation bits. Following this is the listing of the FILE and LABEL source statements.

The instruction text contains machine instructions and data used by the Loader. The first column on the left indicates the octal location of the instruction – note that the example in Figure 9 is taken from a RELMOD assembly and, therefore, the location shown is relative to the load address of the program. Following this is the octal representation of the binary instruction text and Loader control data. The next column contains the binary relocation bits. The next item in the listing is the card image. Finally, at the right-hand side of the page is a single-character position for error flags, and a column containing the statement numbers.

The Literal Pool is placed at the end of this portion of the listing (unless positioned under LITORG control), followed by the EXTERN instructions generated for system symbols (if any) and the END card.

A detailed description of the Loader control data and the relocation bits is provided in the IBM publica-

```
                                        IBMAP ASSEMBLY DKNAME                    11/19/64      PAGE  2
          CONTROL DICTIONARY

                              $IBLDR DKNAME  11/19/64                                          DKNA0001
                              $CDICT DKNAME                                                    DKNA0002

BINARY CARD 00000
 000     000047000007                 PREFACE MODE=REL,PROGRAM-START=00007
         000000000004
 001     244245214425         DKNAME  DKNAME  DECK-START=00000,DECK-END=00047
         000050000000
 002     466463476463         OUTPUT  FILE
         200000000000
 003     255151446227         ERRMSG  EXTERN
         300000000000
 004     665131632560         WRITE   EXTERN
         300000000000
 005     623341673163         S.JXIT  EXTERN
         300000000000
 006     623346472545         S.OPEN  EXTERN
         300000000000
 007     234565255163         CNVERT  ENTRY   AT  00007
         100000000007
 010     255151465160         ERROR   CONTRL  FROM 00032 TO 00035
         100003000032
 011     616160606060         //      CONTRL  FROM 00046 TO 00050
         100002000046

                              $TEXT   DKNAME                                               DKNA0004
```

Figure 8. Control Dictionary Listing

```
BINARY CARD 00000
        000010000002     00011 OUTPUT FILE    U02,*,LRL=14,RCT=1,BLOCK=14,DOUBLE,ERR=ERROR                SUB00000    1
        000000000000     10000
        000000000016     10000
        000001000000     10000
       -010000610200     10000
        011000000016     10000
        000000000000     10000
        020000000000     11100
       -000000000000     10000
                         ERROR  CONTRL  ERROR,ERROR+3                                                      SUB00010    2
                                ENTRY   CNVERT                                                            SUB00020    3
                                EXTERN  ERRMSG,WRITE                                                      SUB00030    4
                                *       LITTLE SUBROUTINE                                                 SUB00040    5
                                        CONTRL  //                                                        SUB00050    6

BINARY CARD 00001
 00046   1 00000 0 00046 00001          USE     //                                                       SUB00060    7
 00046   2 00000 0 00001 00001          BSS     1                                                        SUB00070    8
 00047   2 00000 0 00001 00001 UPPER    BSS     1                                                        SUB00080    9
 00000   1 00000 0 00000 00001          USE     PREVIOUS                                                 SUB00090   10
 00000   000000000067     10000         OCT     67,6667,                                                 SUB00100   11
 00001   000000006667     10000         ETC     666667,66666667                                          SUB00110
 00002   000000666667     10000
 00003   000066666667     10000
 00004   000000000011     10000 NINE     DEC     9                                                       SUB00120   12
 00005  -206060606060     10000 BLANKS   BCI     1,                                                      SUB00130   13
 00006   0 00000 0 00000 10000 SWOPEN   PZE                                                              SUB00140   14
 00007  -1341 07 0 00006 10001 CNVERT   PLT     SWOPEN                                                   SUB00150   15
 00010    0020 00 0 00014 10001          TRA     UPNUM                                                    SUB00160   16
 00011  -1623 06 0 00006 10001          MSM     SWOPEN                                                   SUB00170   17
 00012   0074 00 4 14000 10011          TSX     S.OPEN,4                                                 SUB00180   18
 00013   1 00000 0 04000 10011          PCN     OUTPUT,,0                                                SUB00190   19
 00014  -0500 00 0 22001 10011 UPNUM    CAL     UPPER                                                    SUB00200   20
 00015   0400 00 0 00044 10001          ADD     =1                                                       SUB00210   21
 00016  -1341 05 0 00045 10001          CCS     =012,,5                                                  SUB00220   22

BINARY CARD 00002
 00017   0020 00 0 00030 10001          TRA     RETURN+1                                                 SUB00230   23
 00020   0020 00 0 00030 10001          TRA     RETURN+1                                                 SUB00240   24
 00021   0602 00 0 22001 10011 SETLST   SLW     UPPER                                                    SUB00250   25
 00022   0560 00 0 00005 10001          LDQ     BLANKS                                                   SUB00260   26
 00023  -0765 00 0 00006 10000          LGR     6                                                        SUB00270   27
 00024  -0100 00 0 00023 10001          TNZ     *-1                                                      SUB00280   28
 00025  -0600 00 0 00000 10011          STQ     UPPER-1                                                  SUB00290   29
        0 00001 0 00011 10010
        1 00001 7 00000 10011
 00026  -1627 00 0 10000 10011 PUT      TSL     WRITE                                                    SUB00300   30
 00027   0020 00 4 00000 10000 RETURN   TRA     0,4                                                      SUB00310   31
 00030   0774 00 1 00005 10000          AXT     5,1                                                      SUB00320   32
 00031   2 00001 1 00035 10001          TIX     UPMOR,1,1                                                SUB00330   33
 00032   0774 00 0 00000 10000 ERROR    AXT                                                              SUB00340 W 34
 00033   0074 00 4 06000 10011          TSX     ERRMSG,4                                                 SUB00350   35
 00034   0020 00 0 12000 10011          TRA     S.JXIT                                                   SUB00360   36
 00035   0771 00 0 00006 10000 UPMOR    ARS     6                                                        SUB00370   37
 00036  -1341 05 0 00004 10001          CCS     NINE,,5                                                  SUB00380   38
 00037   0020 00 0 00031 10001          TRA     RETURN+2                                                 SUB00390   39

BINARY CARD 00003
 00040   0020 00 0 00031 10001          TRA     RETURN+2                                                 SUB00400   40
 00041  -0500 00 0 22001 10011          CAL     UPPER                                                    SUB00410   41
 00042   0400 00 1 00004 10001          ADD     NINE,1                                                   SUB00420   42
 00043   0020 00 0 00021 10001          TRA     SETLST                                                   SUB00430   43
                                           LITERALS                                                               44
 00044   000000000001     10000
 00045   000000000012     10000
                                 EXTERN     S.JXIT   GENERATED                                                  47
                                 EXTERN     S.OPEN   GENERATED                                                  48
                  00007 01111    END     CNVERT                                                           SUB00440   49

                  $DKEND DKNAME
```

Figure 9. Assembled Text Listing

tion, *IBM 7040/7044 Operating System (16/32K): Systems Programmer's Guide,* Form C28-6339.

Cross-reference Dictionary Listing

The cross-reference dictionary, as shown in Figure 10, contains four types of information: references to defined symbols, references to location counters, references to multiply defined symbols, and references to undefined symbols.

The listing of references to defincd symbols and multiply-defined symbols gives the symbol, its value, and the statement numbers of those instructions that refer to it.

The P.xxxx symbols are generated by the assembler for literals and with the LITORG and LORG statements.

The listing of references to location counters gives the name of the counter, its starting location, and thc starting and ending statement numbers of the instructions positioned under it.

The listing of references to undefined symbols gives the symbol and the statement number of those instructions that refer to the undefined symbol.

Error Message Listing

Figure 11 illustrates an error message listing. The first column on the left gives the severity level of the error. Following this is the statement number of the instruction in error, the error number, and, finally, the text of the error message.

```
                                    IBMAP ASSEMBLY DKNAME                    11/19/64      PAGE  5
        CROSS-REFERENCE DICTIONARY


REFERENCES TO DEFINED SYMBOLS
 VALUE    NAME      STATEMENT NUMBERS

 00005    BLANKS    26
 00007    CNVERT    49
 VIRTUAL  ERRMSG    35
 00032    ERROR     1
 00004    NINE      38,42
 FILE     OUTPUT    1,19
 00026    PUT
 00027    RETURN    23,24,39,40
 00021    SETLST    43
 VIRTUAL  S.JXIT    36
 VIRTUAL  S.OPEN    18
 00006    SWOPEN    15,17
 00035    UPMOR     33
 00014    UPNUM     16
 00047    UPPER     20,25,29,41
 VIRTUAL  WRITE     30
 00044    P.0000    21
 00045    P.0001    22

REFERENCES TO LOCATION COUNTERS
LC START NAME      STARTING AND ENDING STATEMENT NUMBERS

 00000              1-6,10-49
 00046    //        7-9
```

Figure 10. Cross-reference Dictionary Listing

```
                                    IBMAP ASSEMBLY DKNAME                    11/19/64      PAGE  6
        ERROR MESSAGES

0 STATEMENT 34          ERROR 127   TOO FEW SUBFIELDS IN A STATEMENT

HIGHEST SEVERITY WAS 0.
```

Figure 11. Error Messages Listing

Index

A page number in *italics* indicates that the designed reference is of particular importance.

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601

Printed in U.S.A. C28-6335-2